
STM32Java Platform Architecture

STM32JavaF4 - Keil uVision

User Manual



Reference:	TLT-0596-MAN-STM32JavaF4
Revision:	C
Architecture:	STM32JavaF4
Compiler:	Keil uVision
Product Version:	5.0.2

Confidentiality & Intellectual Property

All right reserved. Information, technical data and tutorials contained in this document are confidential, secret and IS2T S.A. Proprietary under Copyright Law. Without any written permission from IS2T S.A., *copying or sending parts of the document or the entire document by any means to third parties is not permitted* including but not limited to electronic communication, photocopies, mechanical reproduction systems. Granted authorizations for using parts of the document or the entire document do not mean they give public full access rights.

IceTea®, IS2T®, MicroJvm®, MicroEJ®, S3™, SNI™, SOAR®, Drag Emb'Drop™, IceOS®, Shielded Plug™ and all associated logos are trademarks or registered trademarks of IS2T S.A. in France, Europe, United States or others Countries.

Java™ is Sun Microsystems' trademark for a technology for developing application software and deploying it in crossplatform, networked environments. When it is used in this documentation without adding the ™ symbol, it includes implementations of the technology by companies other than Sun.

Java™, all Java-based marks and all related logos are trademarks or registered trademarks of Sun Microsystems Inc, in the United States and other Countries.

Other trademarks are proprietary of their authors.

Table of Contents

1. Bibliography	7
2. Introduction	8
2.1. Scope	8
2.2. Intended Audience	8
2.3. Related Documents	8
2.4. Document Organization	8
3. Platform Development	9
3.1. Introduction	9
3.2. Concepts	9
3.3. Building a Java Platform	11
3.4. Low Level API Pattern	12
3.5. Export a Java Platform	15
4. Application Development	18
4.1. Introduction	18
4.2. MicroEJ Applications	18
4.3. MicroEJ Launches	18
4.4. MicroEJ Tools	20
4.5. Getting Started	21
5. MicroJvm Virtual Machine	32
5.1. Introduction	32
5.2. Functional Description	32
5.3. Architecture	32
5.4. Implementation	33
5.5. Java Language	36
5.6. Java Libraries	36
5.7. Dependencies	37
5.8. Installation	37
5.9. Use	37
6. Native Interface Mechanisms	38
6.1. Simple Native Interface – Green Thread (SNI-GT)	38
6.2. Shielded Plug (SP)	41
6.3. MicroEJ Java H	45
7. Embedded Communications	48
7.1. ECOM	48
7.2. ECOM Comm	49
8. Additional Java Libraries	55
8.1. Native Language Support (NLS)	55
8.2. Logging	57
8.3. Components	58
9. Development Tools	60
9.1. Memory Map Analyzer	60
9.2. Stack Trace Descriptor	62
9.3. Code Coverage Analyzer	64
9.4. Heap Dumper	67
9.5. Test Suite	68
10. Build Support	82
10.1. Board Support Package	82
10.2. Java Examples	83
11. Simulation	84
11.1. Introduction	84
11.2. Functional Description	84
11.3. Mock	85
11.4. Dependencies	88
11.5. Installation	88
11.6. Use	88

12. Launch Options	89
12.1. Category: Libraries	89
12.2. Category: Simulator	96
12.3. Category: Debug	99
12.4. Category: Target	105
13. Appendix	110
13.1. Keil uVision Compiler Compatibility	110
13.2. Floating Point Unit	110
14. Document History	111

List of Figures

3.1. Overall Process	9
3.2. JPF Configuration Overview Tab	10
3.3. JPF Configuration Content Tab	11
3.4. Low Level API Pattern (single implementation)	13
3.5. Low Level API Example	14
3.6. Low Level API Pattern (multiple implementations/instances)	15
4.1. MicroEJ Launch Application Main Tab	19
4.2. MicroEJ Launch Application Execution Tab	19
4.3. JPF Configuration Tab	20
4.4. MicroEJ Tool Configuration	21
4.5. New Project Wizard	22
4.6. New Java Example Wizard First Page	23
4.7. New Java Example Wizard Second Page	23
4.8. MicroEJ Workbench with an Example Project	24
4.9. New Java Project Wizard	25
4.10. New Java Project Wizard First Page	26
4.11. New Java Project Wizard Second Page	27
4.12. New Java Class Wizard	28
4.13. HelloWorld Source Code	29
4.14. Java Project Build Path View	29
4.15. MicroEJ Application Execution	30
4.16. Execution Tab	31
4.17. JPF Configuration Tab	31
5.1. MicroJvm Virtual Machine Flow	32
5.2. A green threads architecture example	33
6.1. SNI Processing	39
6.2. Green threads and RTOS task synchronization	40
6.3. MicroEJ Java H Process	45
7.1. ECOM Flow	48
7.2. ECOM Comm components	50
8.1. Native Language Support Process	55
8.2. Service Oriented Architecture	58
9.1. Memory Map Analyzer Process	60
9.2. Retrieve Map File	61
9.3. Consult Full Memory	61
9.4. Detailed View	62
9.5. Code Coverage Analyzer Process	65
9.6. JUnit final report example	70
9.7. Example Ant - The tree files	71
9.8. Example Ant - Final JUnit report	73
9.9. Example Ant - Final HTML report	73
9.10. Example Java - Available platforms	74
9.11. Example Java - The tree files	75
9.12. Example Java - Main Tab	76
9.13. Example Java - Execution Tab	76
9.14. Example Java - Common Tab	77
9.15. Example Java - Final JUnit report	79
9.16. Example Java - Final HTML report	80
11.1. The HIL connects the SimJPF to the workstation.	84
11.2. A SimJPF socket-connected to its HIL engine.	85
11.3. The SimJPF executes a native Java method foo().	85
11.4. An array and its counterpart in the HIL engine.	86
11.5. Typical usage of HIL engine.	86
11.6. Suspend/resume Java Threads example	86
11.7. GetResourceContent Example	87
11.8. SimJPF Stop Example	87

11.9. Shielded Plug Mock General Architecture	87
---	----

1 Bibliography

[CMREF]	STM32JavaF4 ARMCCv4 Reference Manual (TLT-0595-REF-STM32JavaF4).
[EDC]	Embedded Device Configuration: ESR 021, http://www.e-s-r.net
[B-ON]	Beyond: ESR 001, http://www.e-s-r.net
[SNIGT]	Simple Native Interface for Green Threads: ESR 012, http://www.e-s-r.net
[SP]	Shielded Plug: ESR 014, http://www.e-s-r.net

2 Introduction

2.1 Scope

This document explains how the core features of STM32JavaF4 ARMCCv4 are accessed, configured and used in the MicroEJ workbench. It describes the process for creating and augmenting a Java platform. It also describes how to create a Java application, test it with the simulator, and how the application can inter-operate with C code on the target.

2.2 Intended Audience

The audience for this document is software engineers who need to understand how to create and configure a JPF using the MicroEJ workbench, and how to create, configure and run applications.

2.3 Related Documents

Please refer to [CMREF] for details of the core Java platform components.

2.4 Document Organization

The document is divided into several parts:

- How to develop a new Java platform.
- How to create and run a Java application.
- How the Java platform core works.
- How to create a Java application that can inter-operate with C code.
- How to configure, extend and use the platform (sorted by modules).
- The tooling around the platform.
- How to configure, extend and use the simulator.
- How to use the customize a Java application launcher.

3 Platform Development

3.1 Introduction

This section explains how to create a new Java Platform (usually called platform or JPF), and how to augment it with extra capabilities.

Figure 3.1 shows the overall process. The first 3 steps are performed within the MicroEJ Workbench. The remaining steps are performed within the C IDE.

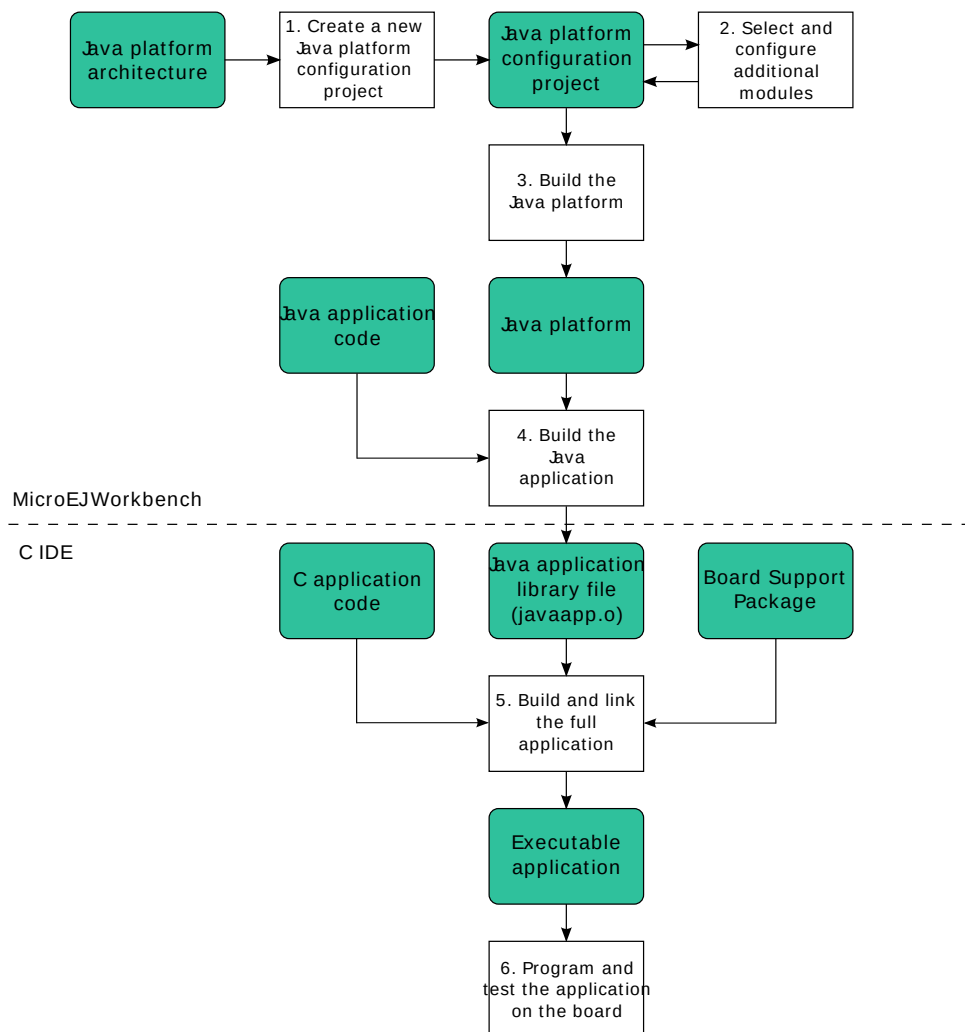


Figure 3.1. Overall Process

3.2 Concepts

3.2.1 Java Platform

A Java platform includes development tools and a runtime environment.

The runtime environment is composed of:

- A MicroJvm Java Virtual Machine.
- Some Java and C libraries.

The development tools is composed of:

- Java APIs to compile Java application code.
- Documentation: user and reference manuals, library specifications, etc.
- Tools: for development and compilation.
- Launch scripts to run the simulation or build the binary file.
- Eclipse plugins.

3.2.2 Java Platform Description

The Java Platforms can be created and configured using a description file. These files are usually called [name].platform and stored at the root of a Java Platform Configuration project called [name]-configuration.

This file is recognized by the MicroEJ Workbench which offers a visualization with two tabs:

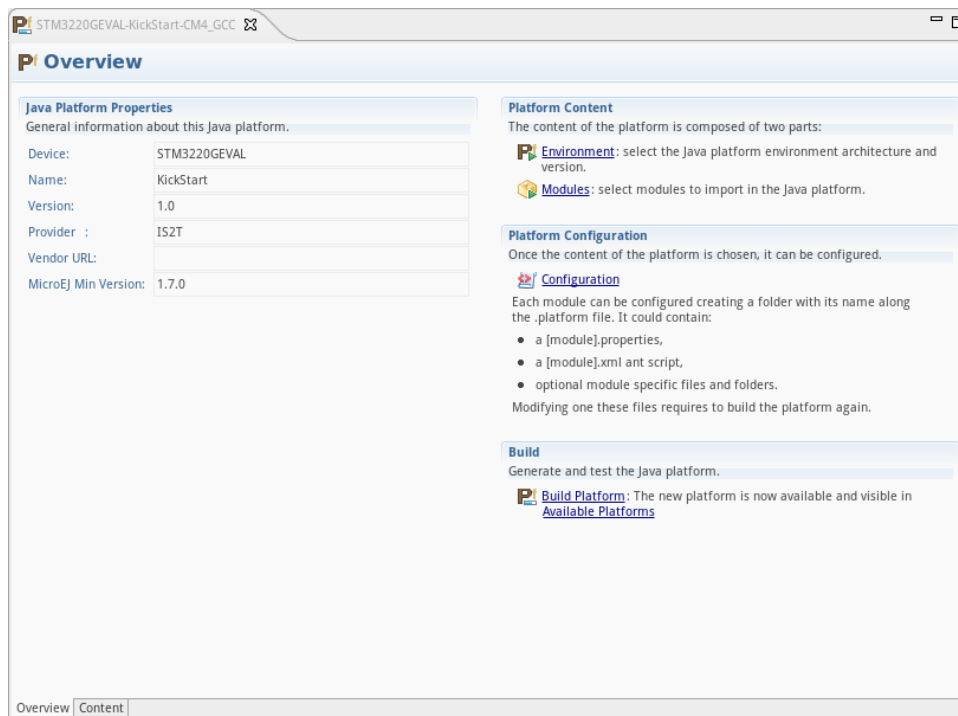


Figure 3.2. JPF Configuration Overview Tab

This tab groups the basic platform information used to identify it: its name, its version etc. These tags can be updated at any time.

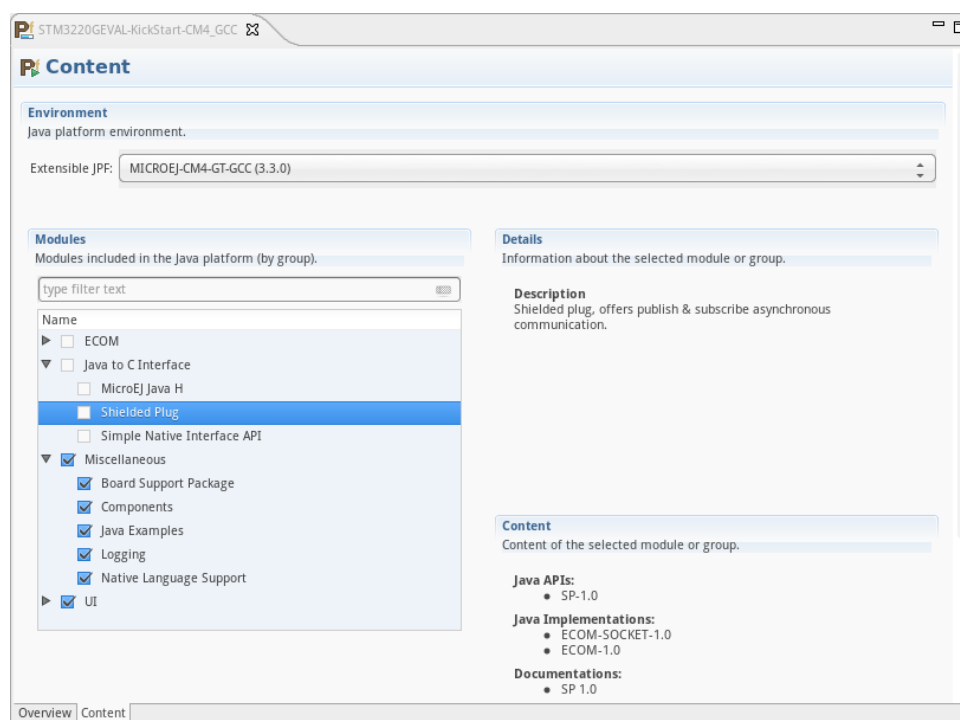


Figure 3.3. JPF Configuration Content Tab

This tab shows all additional modules (see “Modules”) which can be installed into the platform in order to augment its features. The modules are sorted by groups and by functionality. When a module is checked, it will be installed into the platform during the platform creation.

3.2.3 Modules

The primary mechanism for augmenting the capabilities of a “Java Platform” is by adding modules to it.

A MicroEJ module is a group of related files (Java libraries, scripts, link files, native libraries, simulator, tools, etc.) that together provide all or part of a platform capability. Generally, these files serve a common purpose. For example, providing an API, or a library implementation with its associated tools.

The list of modules is in the second tab of the platform configuration tab. A module may require a configuration step to be installed into the platform. The Modules Detail view indicates if a configuration file is required.

3.3 Building a Java Platform

3.3.1 Create a New Java Platform Configuration

The first step is to create a Java Platform configuration:

- Select **File** → **New** → **Project...**, open MicroEJ category and select **Java Platform Configuration**.
- Click on **Next**. The first page allows to set the name of the project.
- Click on **Next**. This page allows to select the basis environment (platform architecture) that contains a minimal Java Platform and a set of compatible modules. This environment could be changed afterwards.

A template can be used by selecting **Create a platform from a template**.

- Click on **Next**. This page contains the identification of the Java Platform to create. Most of the fields have a default value that can be changed.

- Click on **Finish**. A new project is being created containing a [name].platform file.

3.3.2 Groups/Modules Selection

The set of groups of the selected environment is available in the **Modules** tab of the platform description editor.

Each group contains a set of modules. When the group is selected, by default, all its modules are selected. A module can be selected or deselected independently but it is not recommended.

The description and content of an item (group or module) are displayed beside the list when this item is highlighted.

All the checked modules will be installed in the platform.

3.3.3 Platform and Modules Customization

Platform can be customized creating a configuration.xml script beside the [name].platform file. This script can extend one or several of the extension points available.

In the same way, each module can be customized by creating a folder named like the module beside the [name].platform definition. It could contain:

- An optional [module].properties file named after the module name. These properties will be injected in the execution context prefixed by the module name. Some properties might be needed for the configuration of some modules. Please refer to the modules documentation for more information.
- Optional module specific files and folders.

Modifying one these files requires to build the platform again.

3.3.4 Build Java Platform

To build the Java platform, click on the **Build Platform** link on the platform configuration **Overview**.

It will create a Java Platform in the workspace available for the MicroEJ project to run on. The Java Platform will be available alongside the other ones in: **Window** → **Preferences** → **MicroEJ** → **Available platform**.

3.4 Low Level API Pattern

3.4.1 Principle

Each time the user must supply C code that connects a platform component to the target, a *Low Level API* is defined. There is a standard pattern for the implementation of these APIs. Each interface has a name and is specified by two header files:

- [INTERFACE_NAME].h specifies the functions that make up the public API of the implementation. In some cases the user code will never act as a client of the API, and so will never use this file.
- [INTERFACE_NAME]_impl.h specifies the functions that must be coded by the user in the implementation.

The user creates *implementations* of the interfaces, each captured in a separate C source file. In the simplest form of this pattern only one implementation is permitted, as shown in the illustration below.

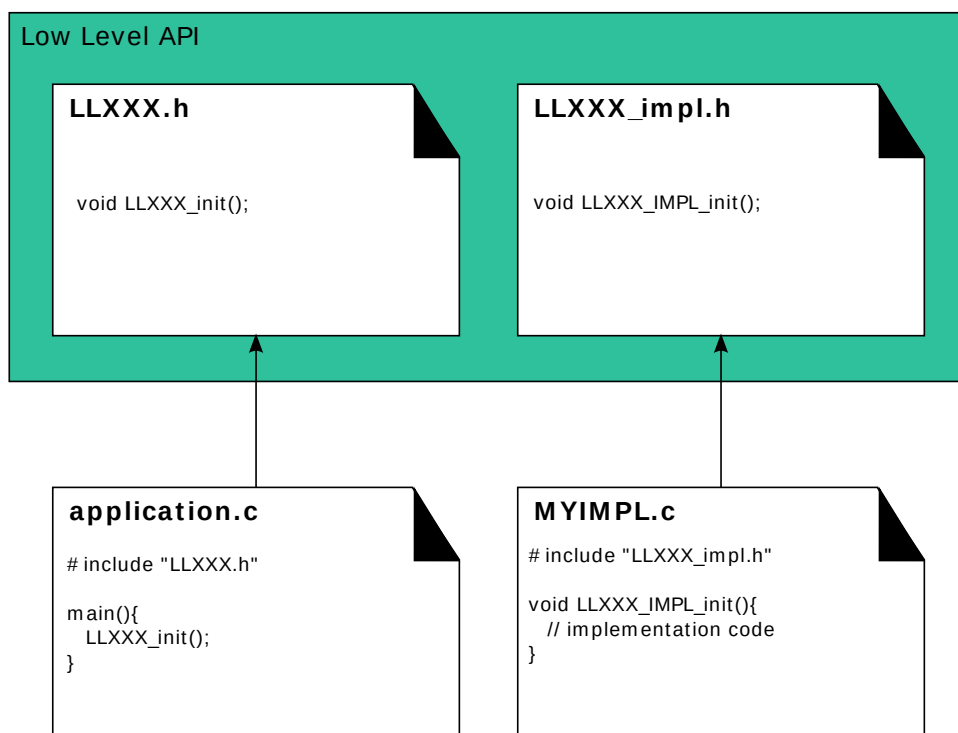


Figure 3.4. Low Level API Pattern (single implementation)

The following figure shows a concrete example of LLAPI. The C world (the board support package) has to implement a send function and has to notify the Java library using a receive function.

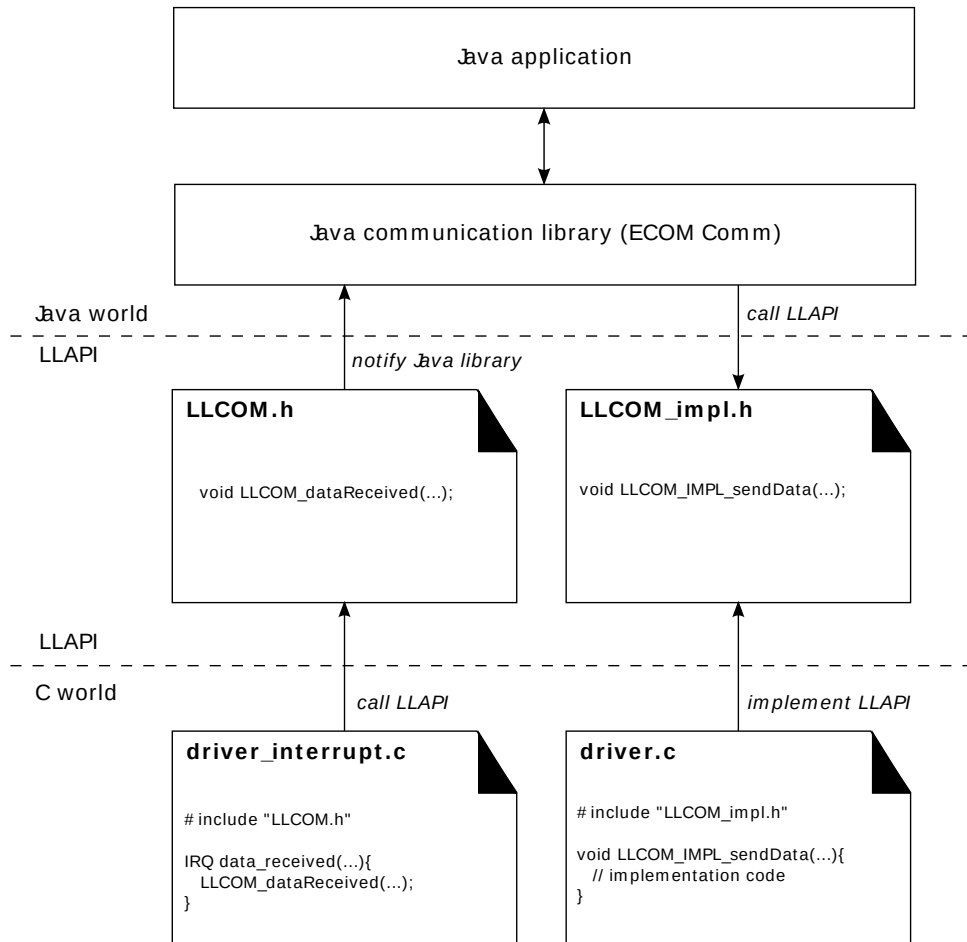


Figure 3.5. Low Level API Example

3.4.2 Multiple Implementations and Instances

When a low level API allows multiple implementations, each implementation must have a unique name. At run-time there may be one or more instances of each implementation, and each instance is represented by a data structure that holds information about the instance. The address of this structure is the handle to the instance, and that address is passed as the first parameter of every call to the implementation.

The illustration below shows this form of the pattern, but with only a single instance of a single implementation.

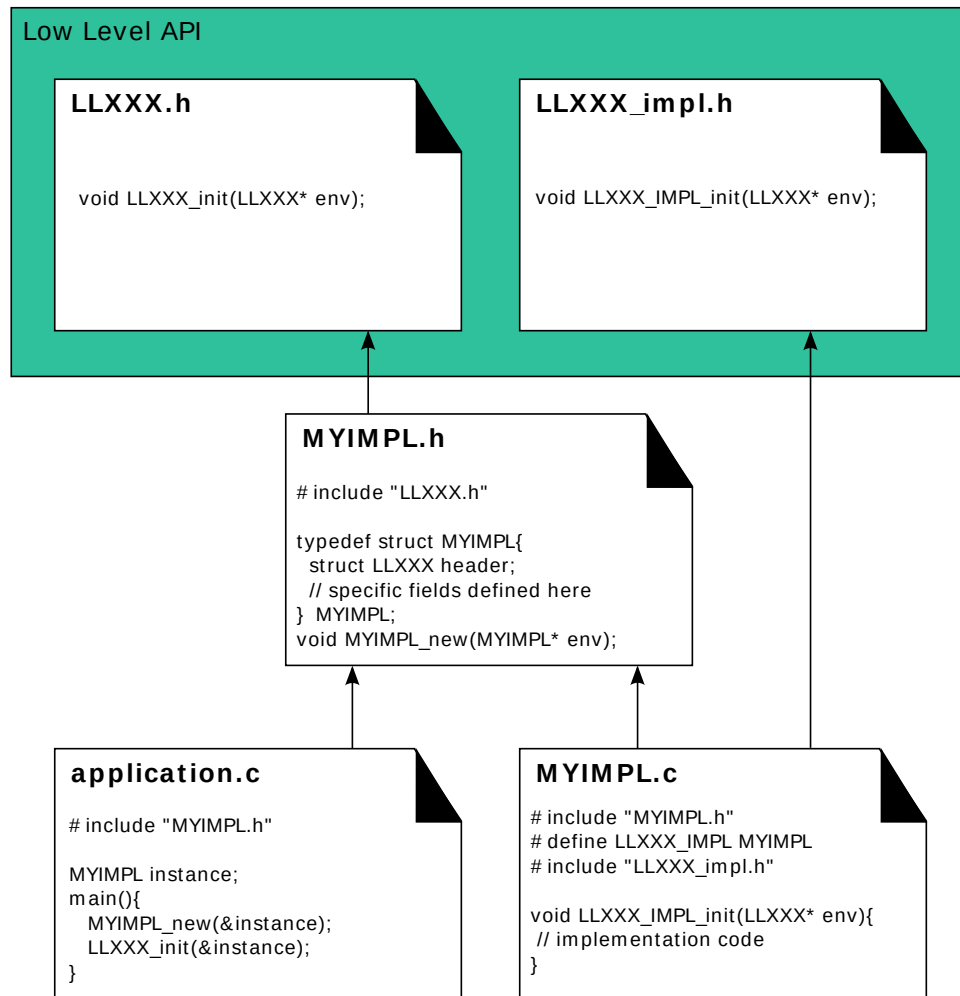


Figure 3.6. Low Level API Pattern (multiple implementations/instances)

The `#define` statement in `MYIMPL.c` specifies the name given to this implementation.

3.5 Export a Java Platform

3.5.1 Principle

As mentioned above a Java platform is divided in two Eclipse projects:

- The Java platform project itself: contains the full simulator engine (SimJPF) and a part of the embedded engine (EmbJPF). This project is created during the Java platform build and should not be modified manually later. Each modification should be described in the Java platform configuration project.
- The board support package project (BSP): contains a third-party C IDE project useful to build and link the final application file. This project references the Java platform project in order to retrieve the Java platform C libraries (MicroJvm virtual machine, communication libraries, etc.).

For the SimJPF, the Java platform project is sufficient. No other project is required to run a Java application on the SimJPF.

The BSP project is required to launch an application on the EmbJPF. The Java application generated by the MicroEJ launch must be linked with the EmbJPF C libraries and the BSP.

Some other projects are available (such as the Java platform configuration project). These projects are required to build the Java platform and there are useless to compile a Java application.

3.5.2 Export the Projects

The simplest way to export a Java platform is to export the BSP project AND the projects required to create the Java platform project. The process consists in importing these projects on a new MicroEJ workspace and then to rebuild the Java platform.

Rebuild the Java platform ensures the update of the links between the local BSP project and the Java platform project.

1. Prepare "export" environment: ensure the configuration project is able to rebuild the Java platform project, the BSP project is ready to link a Java application, etc.
2. Select the Java platform configuration project and click on **File > Export**.
3. In Export window, choose **General > Archive File**.
4. Click on **Next**.
5. Select all projects useful to build the Java platform (Java platform configuration project is already selected) and the BSP project.

Do not select the Java platform project itself. It will be created during the import process of the Java platform.

For the BSP project, do not export the built files (*.o and *.a files): these files will be built again later.

6. Choose an output location and the name for the zip file which will contain the exported projects.
7. Click on **Finish**. The zip file contains all the selected projects.

Follow the next steps to import the Java platform zip file in another workspace:

1. Use the same version of MicroEJ to ensure that the Java platform architecture and additional extensions are installed into the MicroEJ repository.
2. **File > Import**.
3. Select sub menu **General > Existing Projects into Workspace**.
4. Click on **Next**.
5. Check **Select archive file** and browse to retrieve the zip file.
6. Select only the project previously exported. Some extra projects may appear when the folder into a project contains an Eclipse .project file.
7. Click on **Finish**. The projects are now imported into the new workspace.
8. Open the Java platform configuration file (available in the Java platform configuration project).
9. Build the Java platform. A new Java platform project is now available. The links between the BSP and the Java platform are automatically updated.
10. Build a Java application against the new Java platform.
11. Open the third-party C IDE, build and link the final application.

3.5.3 Export a .jpf File

Another way to export a Java platform consists in exporting the platform project instead of the configuration project. The result is a .jpf file that can be imported by the MicroEJ workbench.

This procedure can be used to export easily the simulation part (SimJPF) of the Java platform. The embJPF of the resulting platform may not be usable standalone. A third-party C project (BSP project) may be required to get a fully consistent platform.

To have more information on this export way please consult the workbench documentation: [Help > Help Contents > MicroEJ Platform Developer Guide](#).

4 Application Development

4.1 Introduction

The MicroEJ workbench provides the ability to:

- develop Java applications,
- launch Java applications on a board (platform embedded side),
- launch Java applications on the simulator (platform simulation side),
- launch some tools related to a platform.

4.2 MicroEJ Applications

MicroEJ applications are developed as standard Java applications on Eclipse JDT, using MicroEJ libraries. MicroEJ workbench allows to run / debug / deploy MicroEJ applications on Java Platforms.

Section “Trying Out Examples” describes how to import a ready to use MicroEJ Example project.

Section “Writing a First MicroEJ Application” describes how to create a MicroEJ application project from scratch.

4.3 MicroEJ Launches

The MicroEJ launch configuration sets up the “MicroEJ Applications” environment (main class, resources, target platform, platform specific options) and then launches a MicroEJ launch script for execution.

Execution is done either on SimpJPF (desktop simulation) or on EmbJPF (hardware target). The launch operation is platform specific. It may depend on external tools the platform requires (such as target memory programming). Refer to the platform specific documentation for more informations about the available launch settings.

See section “Launching a MicroEJ Application” about the launches creation.

4.3.1 Main Tab

The Main tab allows to set in order:

1. The main project of the application.
2. The main class of the application containing the main method.
3. Types required in your application, that are not statically embedded from the main class entry point. Most required types are those that may be loaded dynamically by the application using `Class.forName()` method.
4. Binary resources that need to be embedded by the application. Most often loaded by the application using `Class.getResourceAsStream()` method.
5. Immutable objects description files. See [B-ON 1.2] ESR documentation for use of immutable objects.

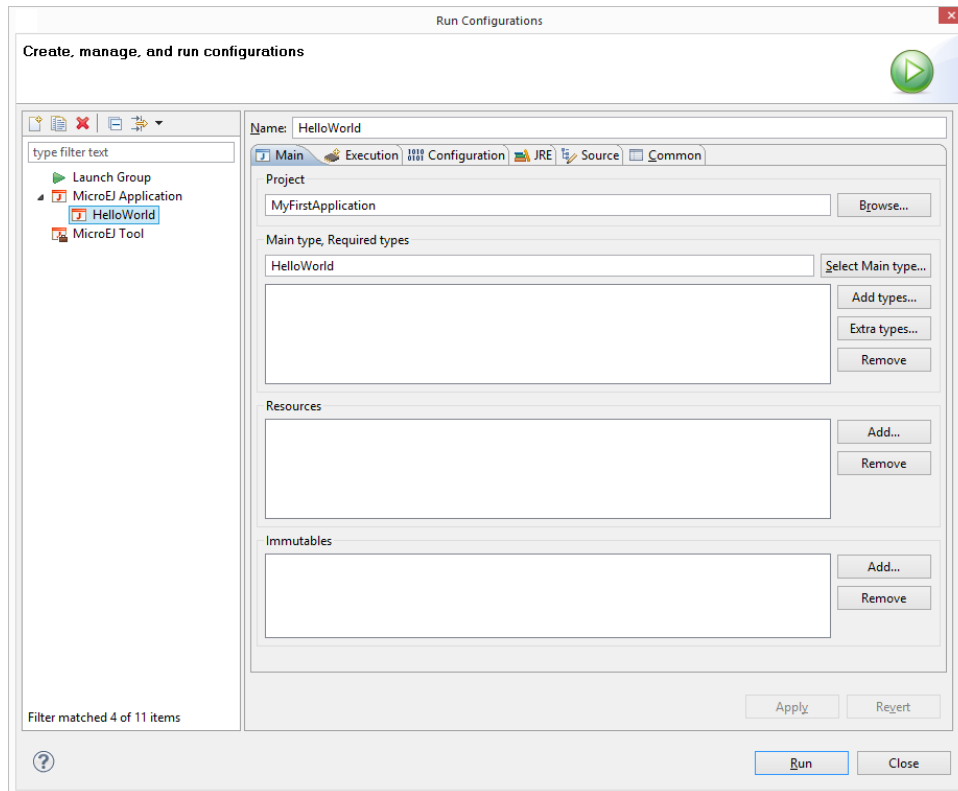


Figure 4.1. MicroEJ Launch Application Main Tab

4.3.2 Execution Tab

Next tab is the Execution tab. Here the target needs to be selected. Choose between execution on SimJPF (desktop simulation) or on EmbJPF (hardware target). Each of them may provide multiple launch settings. Moreover, on EmbJPF execution, Platforms may provide multiple MicroJVM operating modes depending on board debug features (trace, embedded symbolic debug, ...). This page also allows to keep generated intermediate files and to print verbose options (advanced debug purpose options).

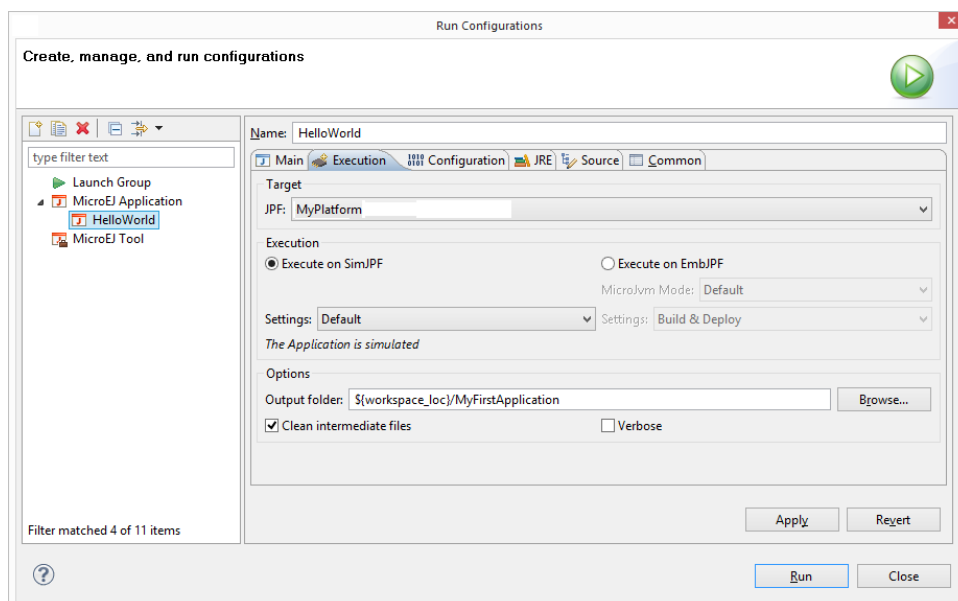


Figure 4.2. MicroEJ Launch Application Execution Tab

4.3.3 JPF Configuration Tab

Next tab is the JPF Configuration tab. This tab contains all platform specific options.

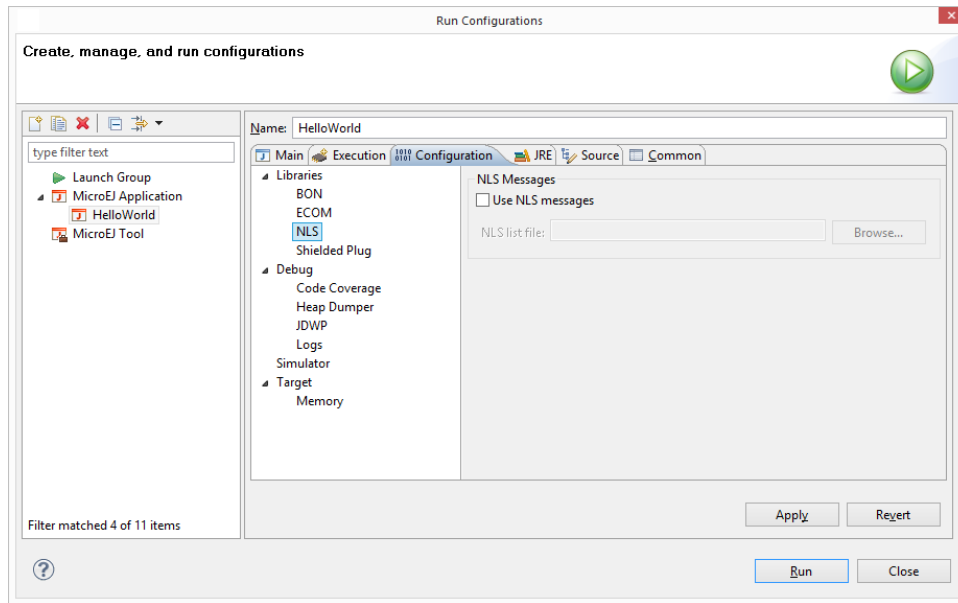


Figure 4.3. JPF Configuration Tab

4.3.4 JRE Tab

Next tab is the JRE tab. This tab allows to configure the Java Runtime Environment used for running the underlying launch script. It does not configure the MicroEJ application execution. The `VM Arguments` text field allow to set vm specific options, most likely for increasing memory spaces:

- to modify heap space to 1024MB, set `-Xmx1024M` option,
- to modify string space (also called PermGen space) to 256MB, set `-XX:PermSize=256M -XX:MaxPermSize=256M` options,
- to set threads stacks space to 512MB, set `-Xss512M` option

4.3.5 Other Tabs

Next tabs (Source and Common tabs) are the default Eclipse launch tabs. Refer to Eclipse help for more details on how to use these launch tabs.

4.4 MicroEJ Tools

A JPF project contains a number of tools to assist with various aspects of development. Some of these tools are run using **MicroEJ Tool** configurations, created using the **Run Configurations** dialog of the workbench. A configuration must be created for the tool before it can be used.

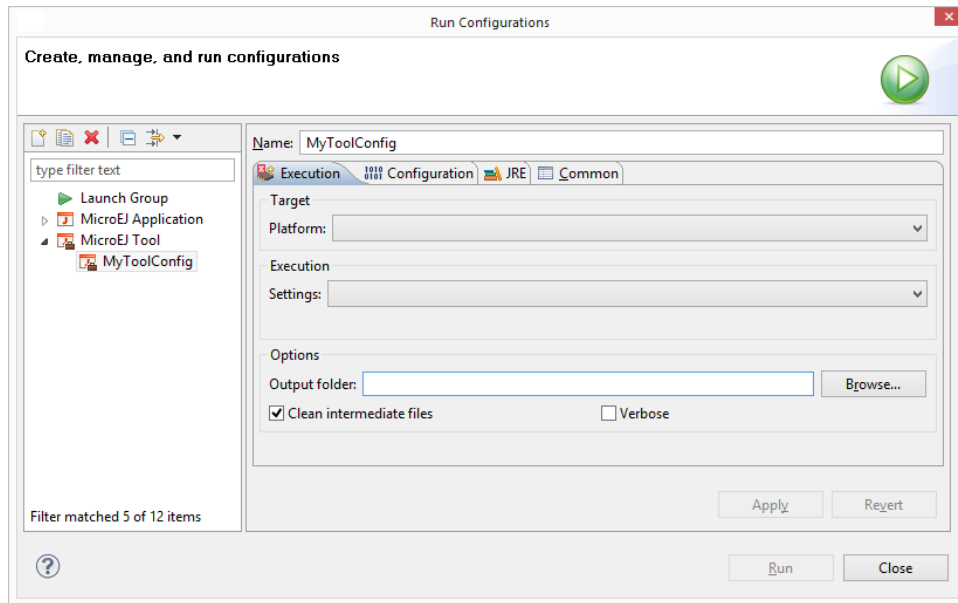


Figure 4.4. MicroEJ Tool Configuration

Figure 4.4 shows a tool configuration being created. The JPF has been selected but the selection of which tool to run has not yet been made. That selection is made in the **Execution Settings...** box. The **JPF Configuration** tab then contains the options relevant to the selected tool.

4.5 Getting Started

4.5.1 Trying Out Examples

Platforms usually come with a set of examples. A MicroEJ example is a MicroEJ project with source code, configured launchers and additional documentation.

- Select **File** → **New** → **Example...**, open MicroEJ category and select Java Example. A shortcut alternative is to open the MicroEJ perspective (**Window** → **Open perspective** → **MicroEJ**) and then select **File** → **New** → **Java Example**.

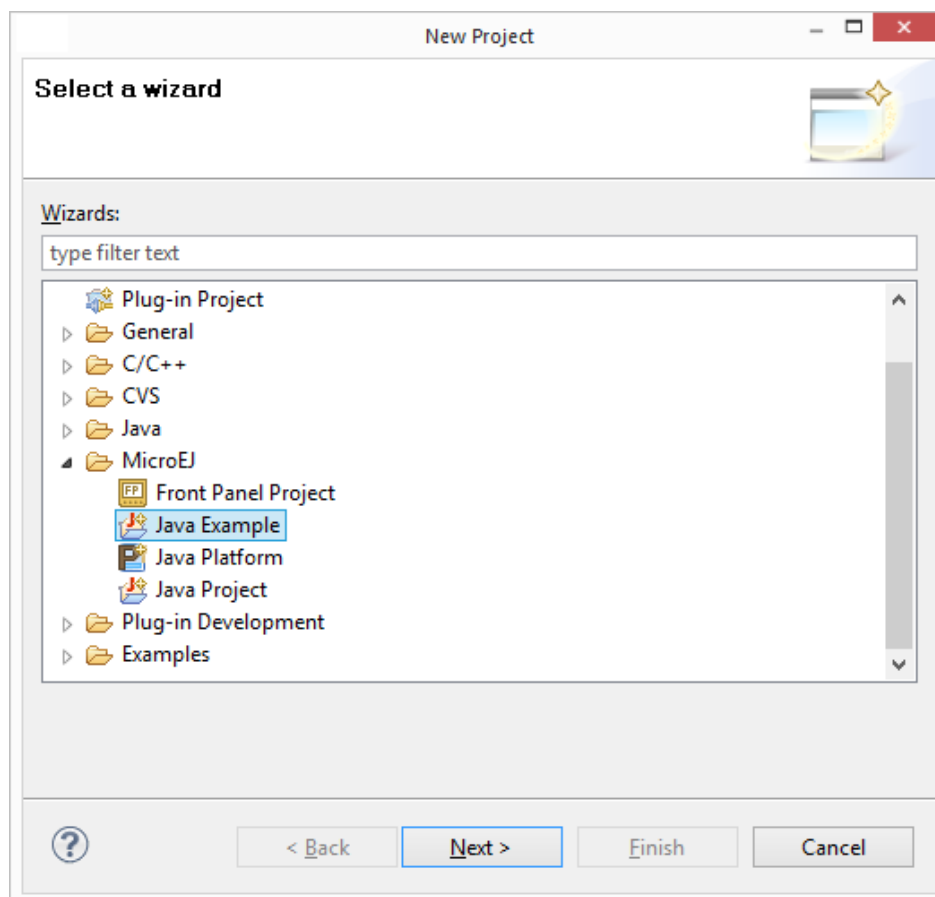


Figure 4.5. New Project Wizard

- Click on **Next...** to open the example selection page. When a target platform is selected, the list of all available examples is displayed. Examples are organized according to their categories (Application Notes, samples, Demos, ...), and according to platform specific categories. When an example is selected, a short description is displayed at the bottom of the page.

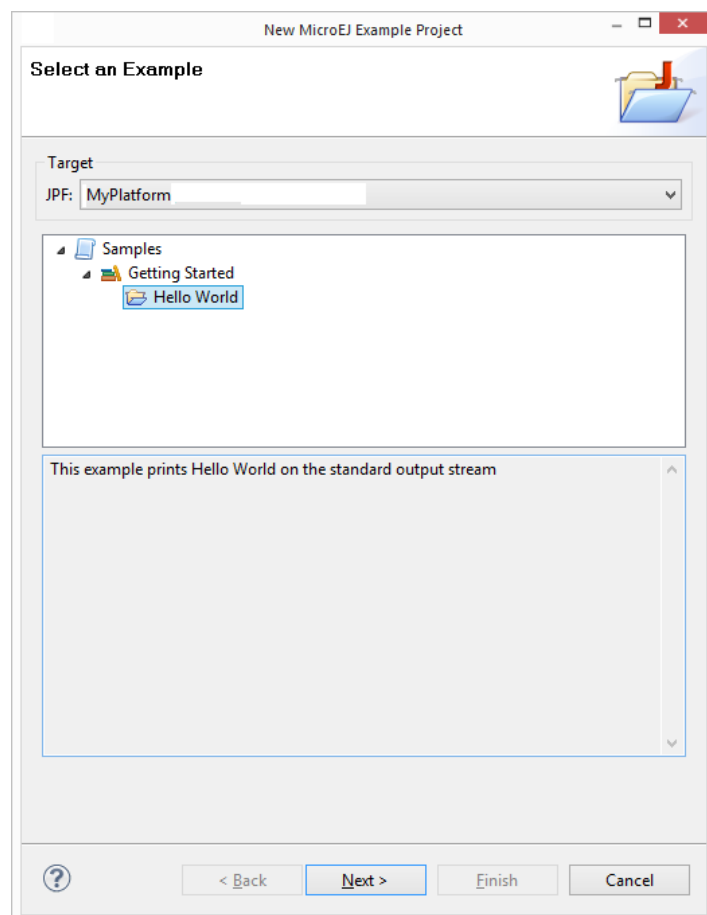


Figure 4.6. New Java Example Wizard First Page

- Select an example and click on **Next**. The following page prompts the name of the project being created (a default name is proposed).

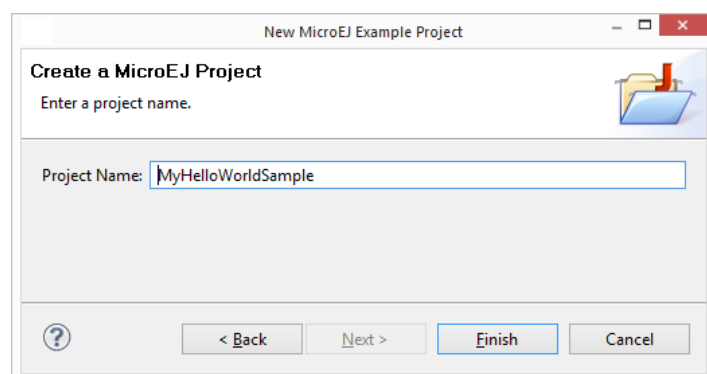


Figure 4.7. New Java Example Wizard Second Page

- Click on **Finish**. The selected example is imported into a project with given name. The main class and/or associated documentation are automatically opened. Documentation is stored in documentation directory.

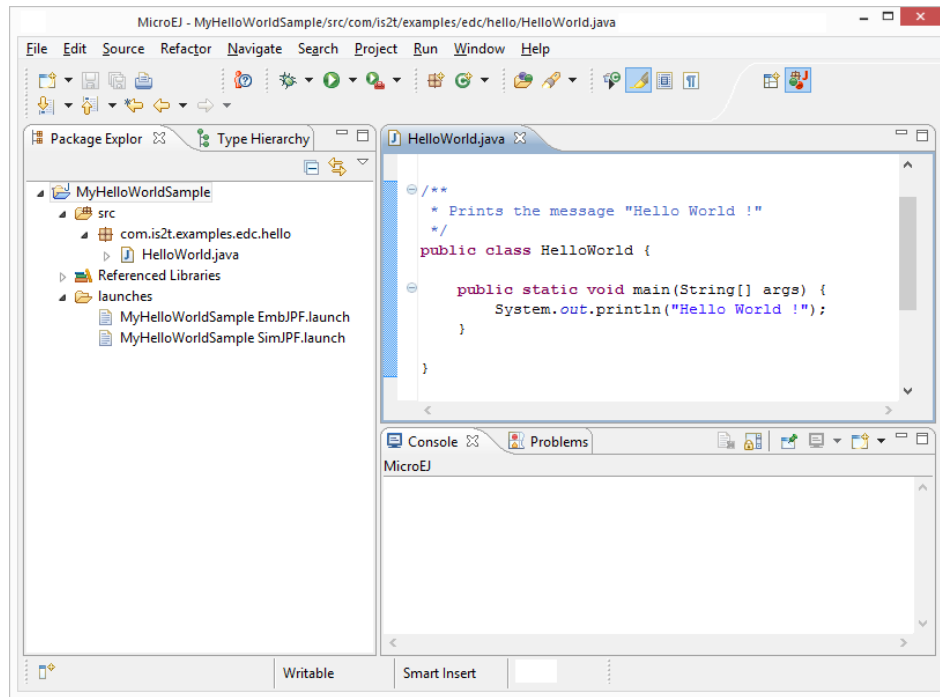


Figure 4.8. MicroEJ Workbench with an Example Project

Once the program is compiled, it is possible to launch and run it on a platform (see “Launching a MicroEJ Application”). MicroEJ launchers are available in the **launches** directory.

4.5.2 Writing a First MicroEJ Application

This section describes all necessary steps to develop a simple `HelloWorld` MicroEJ application. No Java/Eclipse skills are required.

4.5.2.1 Creating a New MicroEJ Project

Before writing the application code, you need to create a MicroEJ project named `myFirstApplication`.

- Select **File** → **New** → **Project...**, open MicroEJ category and select **Java Project**. A shortcut alternative is to open the MicroEJ perspective (**Window** → **Open perspective** → **MicroEJ**) and then select **File** → **New** → **Java Project**.

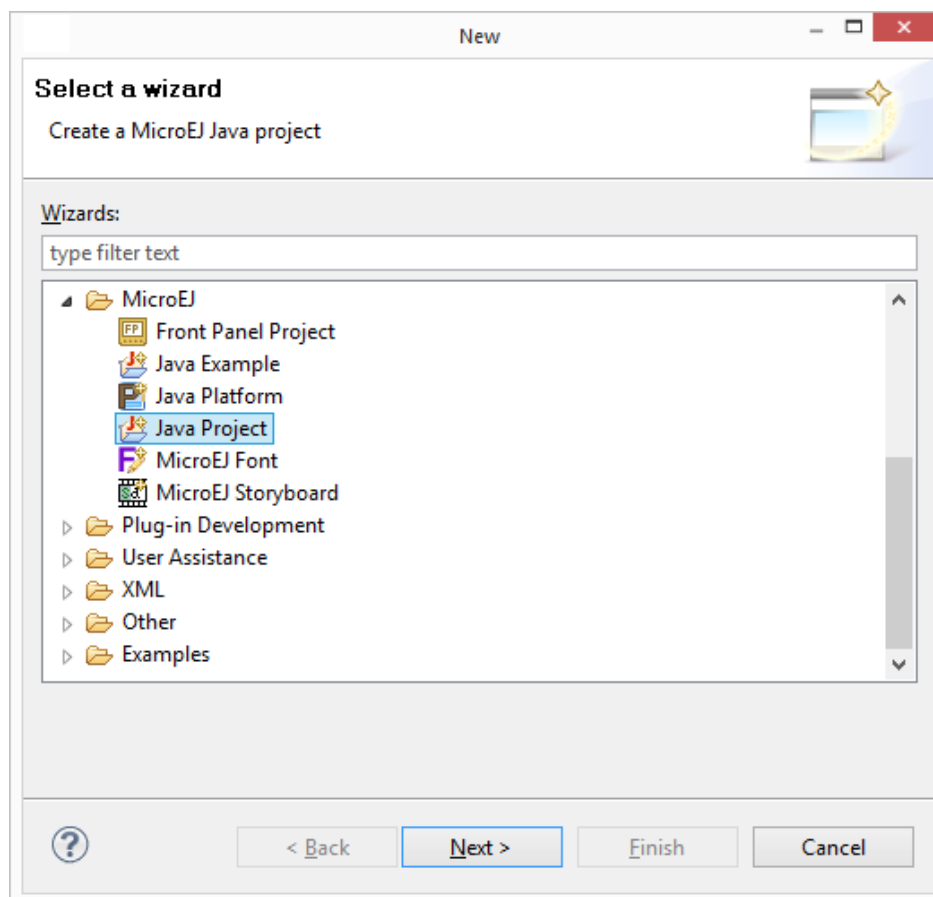


Figure 4.9. New Java Project Wizard

- Click on **Next**. The following page allows to set the name of the project and the list of involved libraries. The set of available libraries depends on the installed platforms. By default, a minimal configuration is required (for example EDC-1.2 that is selected by default).

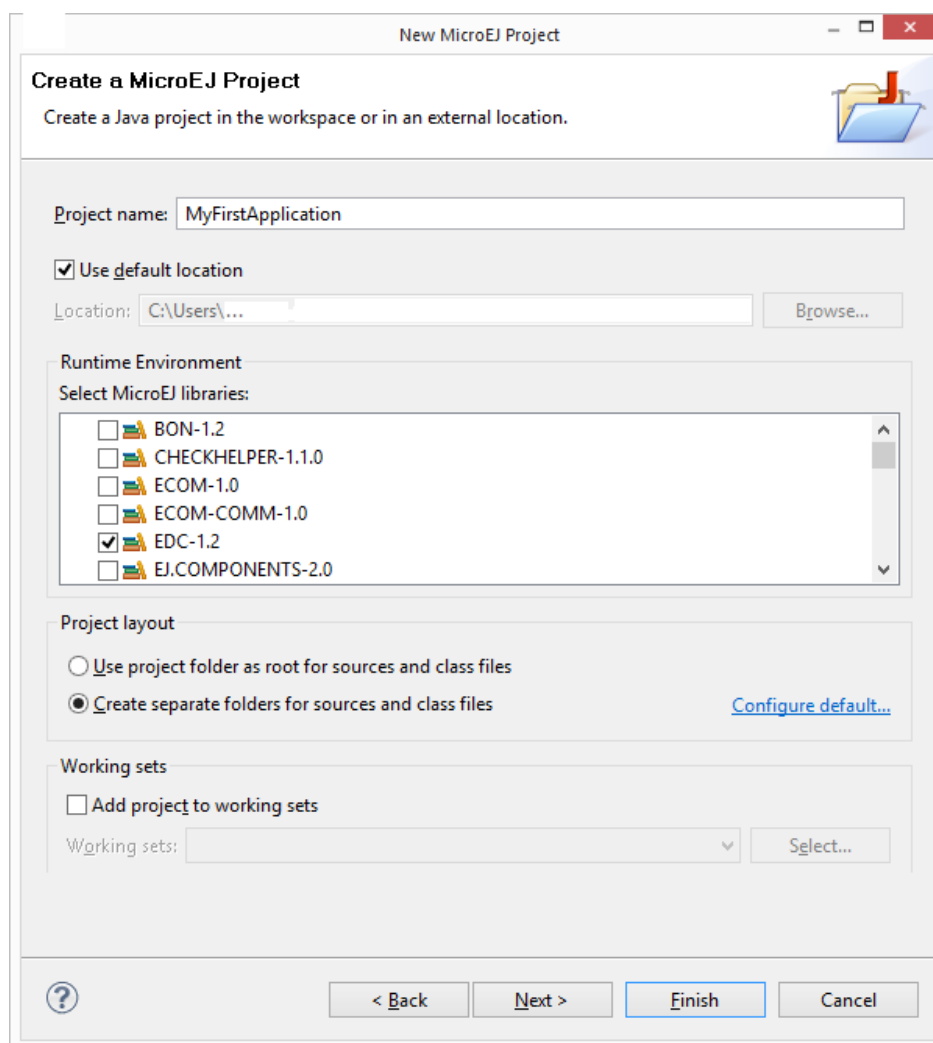


Figure 4.10. New Java Project Wizard First Page

- Click on **Next**. The following page is the standard Eclipse JDT wizard page that allows to reference projects/additional libraries, and to manage classpaths order. For this tutorial, the page is left unchanged. For additional information, refer to the Eclipse JDT documentation.

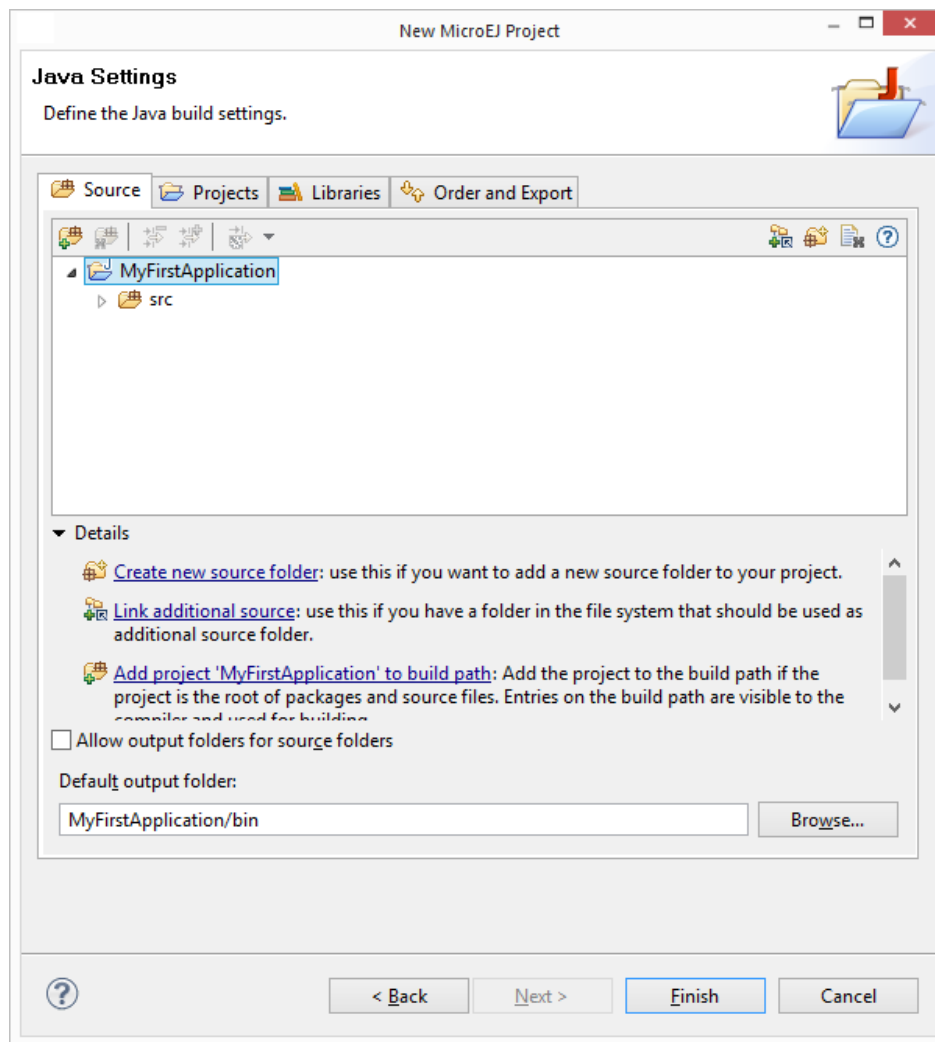


Figure 4.11. New Java Project Wizard Second Page

- Click on **Finish**. The project is being created in the **Package Explorer**.

4.5.2.2 Creating a New Java Class

Once a MicroEJ project is created, you are ready to develop a Java class.

- Right-click on the source folder `src` and select **New** → **Class**. The JDT wizard asks for a class name. For this tutorial, the class name is `HelloWorld`. Leave other options unchanged. For more information refer to the Eclipse JDT documentation.

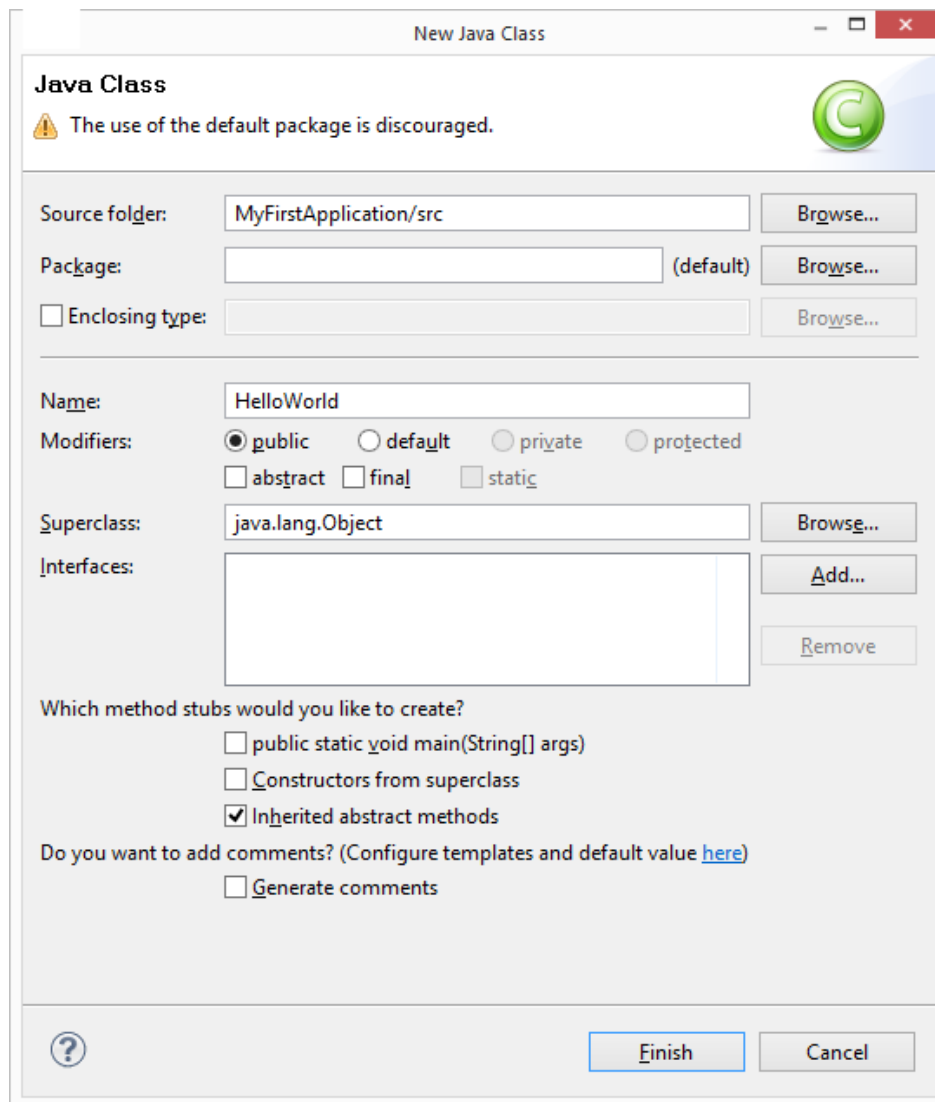


Figure 4.12. New Java Class Wizard

- Then click on Finish. A new file HelloWorld.java is created in the src folder. By default it has created the class skeleton. Replace the contents by the following code. Then save the file (**New** → **Save**). The class is compiled automatically by Eclipse JDT.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
        System.out.println("This is my first MicroEJ application!");
    }
}
```

Example 4.1. HelloWorld.java

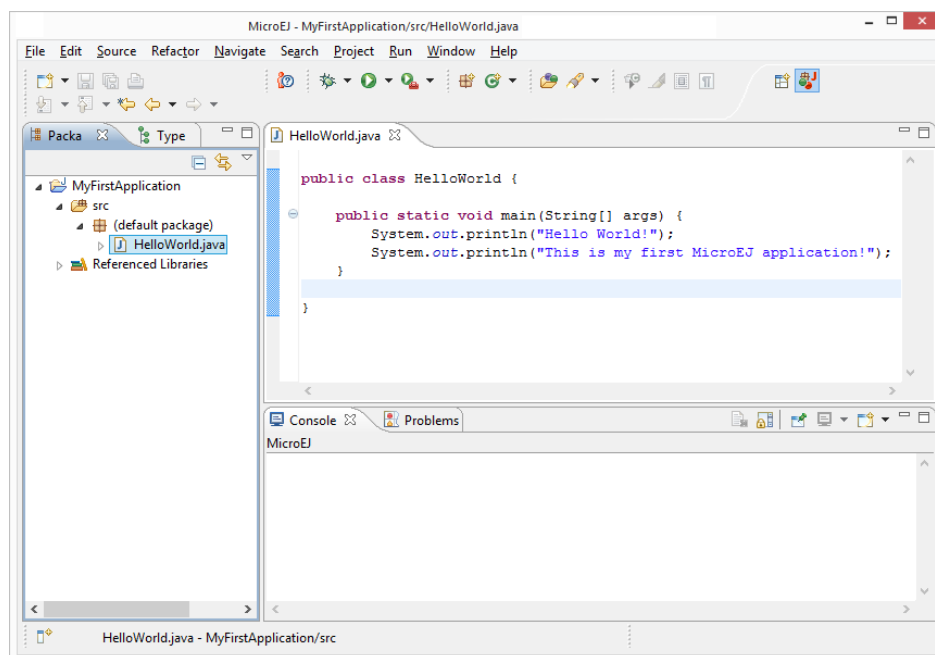


Figure 4.13. HelloWorld Source Code

Once the program is written and compiled, it is possible to launch and run it on a platform (see “Launching a MicroEJ Application”).

4.5.2.3 Update Project Libraries Dependencies

Once a MicroEJ project is created (see section “Writing a First MicroEJ Application”), its classpath dependencies can be updated at any time. In project explorer view, right-click on project **Properties** → **Java Build Path**. MicroEJ libraries dependencies are configured in the Libraries tab view.

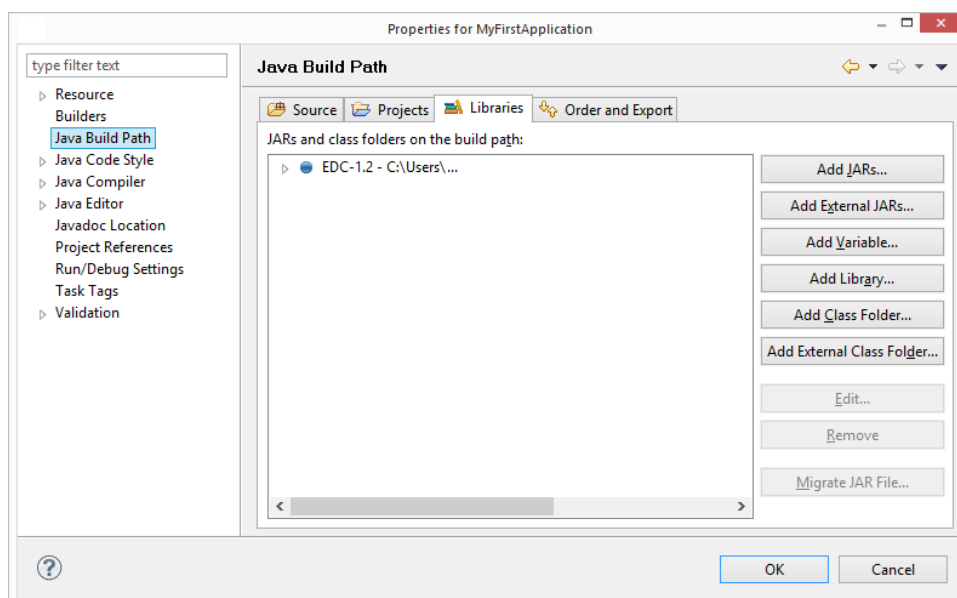


Figure 4.14. Java Project Build Path View

A MicroEJ Library is referenced by an Eclipse classpath variable which is automatically managed by MicroEJ Workbench. To choose a new MicroEJ Library, click on **Add Variable...** button and select the classpath variable with referring to the required library name and version.

4.5.3 Launching a MicroEJ Application

This section explains how to configure a new MicroEJ application launch for a Java Platform. The tutorial uses the HelloWorld application example described in “Writing a First MicroEJ Application”.

Edit the main class (i.e. the class that holds a Java main method) of your application. For example, select **Navigate** → **Open Type** → **HelloWorld** → **OK**. Open the run dialog box (**Run** → **Run configurations...**) and double-click on MicroEJ Application launch type. A new launch named HelloWorld is created. Project and main type are automatically filled according to the previously selected class (HelloWorld). Main tab has no errors.



Warning

If the project or the class has changed, the Launch tab is not updated. You should manually select a valid project and main type.

When all fields are filled with a valid value, the Run button is enabled. Click on Apply to validate. Click on Run to start it.

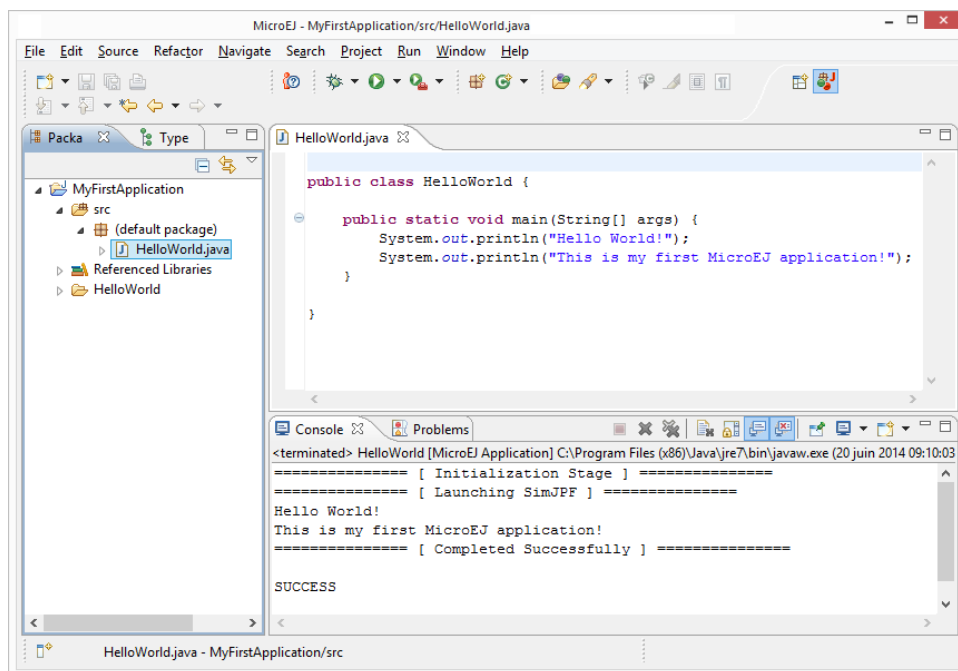


Figure 4.15. MicroEJ Application Execution

4.5.4 Building and Running the Application for Embedding

4.5.4.1 Building the MicroEJ Application for Embedding

To run the application on the target you must first build the Java application. To do this, create a launch configuration and select **Execute on EmbJPF**, as shown below:

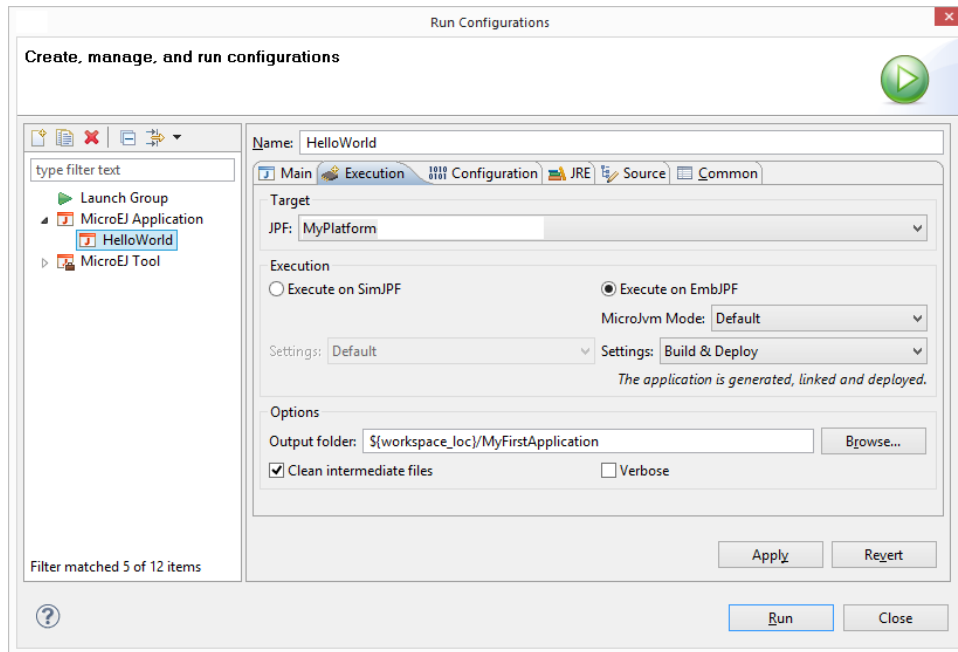


Figure 4.16. Execution Tab

The **JPF Configuration** tab has an option that specifies where the built Java application should be put. It is usual to put it in the board support package BSP libraries folder, and calling it `javapp.o`:

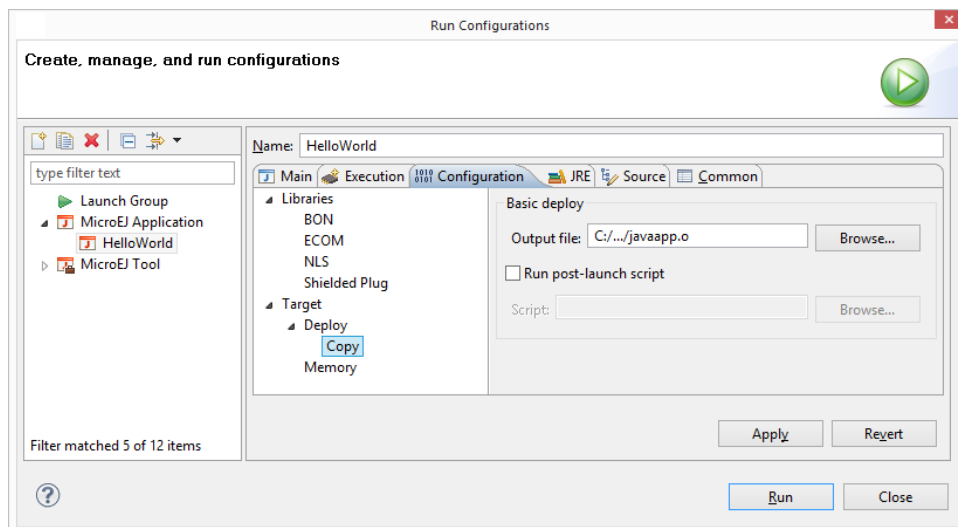


Figure 4.17. JPF Configuration Tab

With these settings, the output file will be called `javapp.o`.

4.5.4.2 Linking the Java Platform

To run the application, the Java application (`javapp.o` in the example above) must be linked with user-supplied drivers, other user-supplied C code and the libraries found in the following work-in-progress JPF directories:

- `source/MICROJVM/lib`
- `source/lib`

5 MicroJvm Virtual Machine

5.1 Introduction

The MicroJvm Virtual Machine (also called platform engine) and its components represent the core of the platform. It is used to compile and execute at runtime the Java application code.

5.2 Functional Description

Figure 5.1 shows the overall process. The first two steps are performed within the MicroEJ Workbench. The remaining steps are performed within the C IDE.

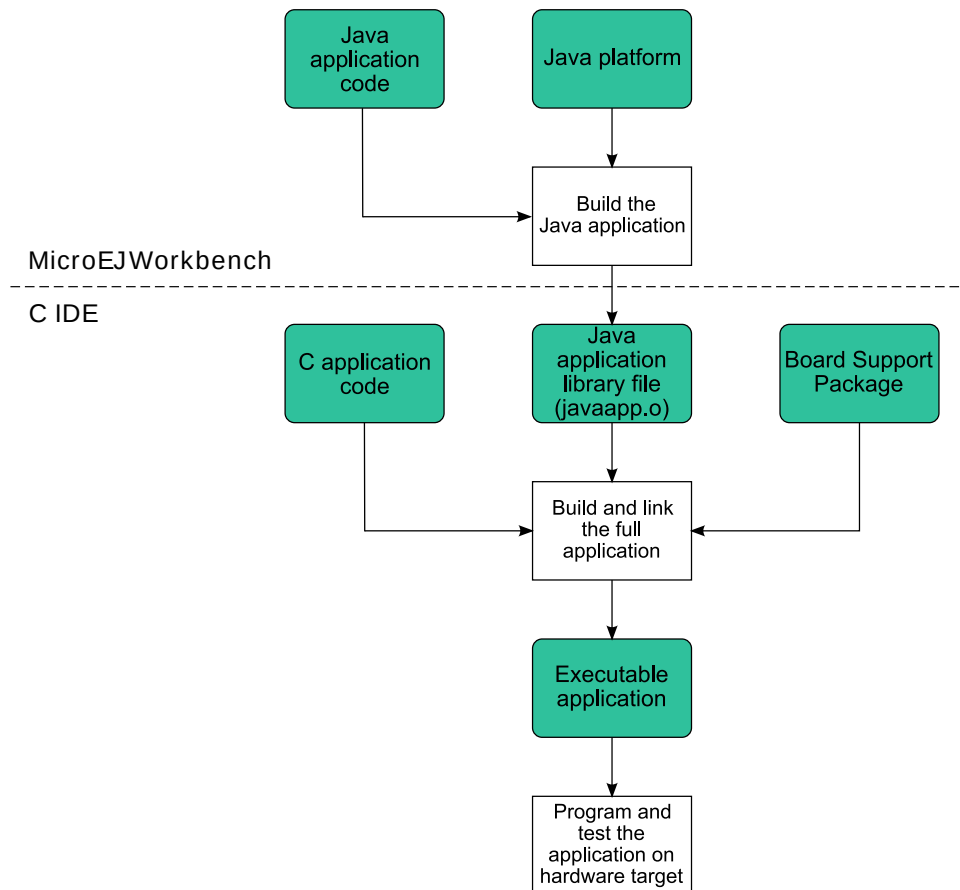


Figure 5.1. MicroJvm Virtual Machine Flow

1. Step 1 consists in writing a Java application against a set of Java libraries available in the platform.
2. Step 2 consists in compiling the Java application code and the required Java libraries in an ELF library using the Smart Linker (SOAR) .
3. Step 3 consists in linking the previous ELF file with the MicroJvm Virtual Machine library and a 3rd-party BSP (OS, drivers etc.). This step may require a third party linker provided by a C toolchain.

5.3 Architecture

The MicroJvm Virtual Machine and its components have been compiled for one specific CPU architecture and with a specific C compiler.

The architecture of the platform engine is called green thread architecture, it runs in a single RTOS task. Its behavior consists in scheduling Java threads. The scheduler implements a priority preemptive sched-

uling policy with round robin for the Java threads with the same priority. In the following explanations the term "RTOS task" refers to the tasks scheduled by the underlying OS and the term "Java thread" refers to the thread scheduled by the MicroJVM Virtual Machine.

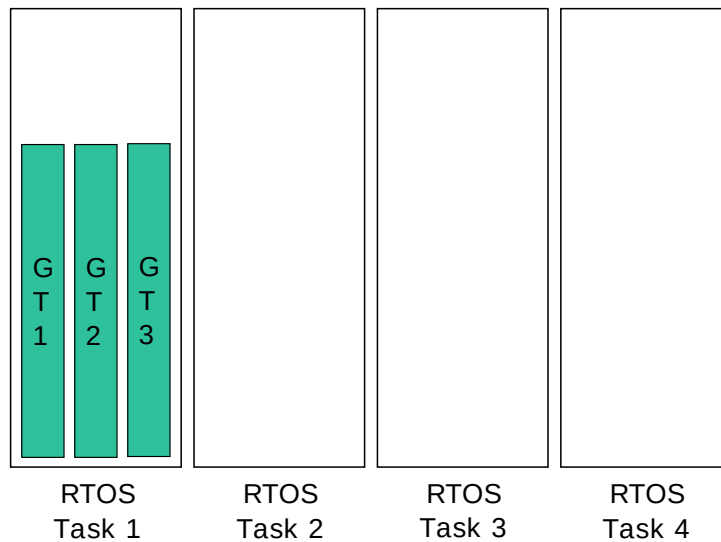


Figure 5.2. A green threads architecture example

The activity of the platform is defined by the Java application. When the Java application is blocked (all Java threads are sleeping), the platform sleeps entirely: the RTOS task that runs the platform sleeps.

The platform is responsible for providing the time to the Java world: the precision is 1 millisecond.

5.4 Implementation

The platform implements the [SNIGT] specification. It is created and initialized with the C function `SNI_createVM`. Then it is started and executed in the current RTOS task by calling `SNI_startVM`. The function `SNI_startVM` returns when the Java application exits. The function `SNI_destroyVM` handles the platform termination.

The file `LLMJVM_impl.h` that comes with the platform defines the API to be implemented. The file `LLMJVM.h` that comes with the platform defines platform specific exit codes constants.

5.4.1 Initialization

The Low Level MicroJVM API deals with two objects: the structure that represents the platform, and the RTOS task that runs the platform. Two callbacks allow engineers to interact with the initialization of both objects:

- `LLMJVM_IMPL_initialize` : called once the structure representing the platform is initialized.
- `LLMJVM_IMPL_vmTaskStarted` : called when the platform starts its execution. This function is called within the RTOS task of the platform.

5.4.2 Scheduling

To support the green thread round-robin policy, the platform assumes there is a RTOS timer or some other mechanism that counts (down) and fires a call-back when it reaches a specified value. The platform initializes the timer using the `LLMJVM_IMPL_scheduleRequest` function with one argument: the absolute time at which the timer should fired. When the timer fires, it must call the `LLMJVM_schedule` function, which tells the platform to execute a green thread context switch (gives another Java thread a chance to run).

5.4.3 Idle Mode

When the platform has no Java activity to execute, it calls the `LLMJVM_IMPL_idleVM` function, which is assumed to put in sleep state the RTOS task of the platform. `LLMJVM_IMPL_wakeupVM` is called to wake up the platform task. When the platform task really starts to execute again, it calls the `LLMJVM_IMPL_ackWakeup` function to acknowledge its restart of activity.

5.4.4 Time

The platform defines two times:

- the application time: the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC.
- the system time: the time since the start of the device. This time is independent of any user considerations and cannot be set.

The platform relies on the following C functions to provide those times to the Java world:

- `LLMJVM_IMPL_getCurrentTime` : depending on the parameter (`true / false`) must return the application time or the system time. This function is called by the Java method `System.currentTimeMillis()`. It is also used by the platform scheduler and should be implemented efficiently.
- `LLMJVM_IMPL_getTimeNanos` : must return the system time in nanoseconds.
- `LLMJVM_IMPL_setApplicationTime` : must set the difference between the current time and midnight, January 1, 1970 UTC.

5.4.5 Example

The following example shows how to create and launch the MicroJVM Virtual Machine from the C world. This function (`microjvm_main`) should be called from a dedicated RTOS task.

```

#include <stdio.h>
#include "microjvm_main.h"
#include "LLMJVM.h"
#include "sni.h"

void microjvm_main(void)
{
    void* vm;
    int32_t err;
    int32_t exitcode;

    // create VM
    vm = SNI_createVM();

    if(vm == NULL)
    {
        printf("VM initialization error.\n");
    }
    else
    {
        printf("VM START\n");
        err = SNI_startVM(vm, 0, NULL);

        if(err < 0)
        {
            // Error occurred
            if(err == LLMJVM_E_EVAL_LIMIT)
            {
                printf("Evaluation limits reached.\n");
            }
            else
            {
                printf("VM execution error (err = %d).\n", err);
            }
        }
        else
        {
            // VM execution ends normally
            exitcode = SNI_getExitCode(vm);
            printf("VM END (exit code = %d)\n", exitcode);
        }

        // delete VM
        SNI_destroyVM(vm);
    }
}

```

Example 5.1. MicroJvm Virtual Machine Creation

5.4.6 Debugging

The internal MicroJvm Virtual Machine function called `LLMJVM_dump` allows to dump the state of all the Java threads: name, priority, stack trace, etc. This function can be called at any time and from an interrupt routine (for instance from a button interrupt).

This is an example of a dump:

```

===== VM Dump =====
2 java threads
-----
Java Thread[3]
name="SYSINpmp" prio=5 state=WAITING

java/lang/Thread:
  at com/is2t/microbsp/microui/natives/NSystemInputPump.@134261800
  [0x0800AC32]
  at com/is2t/microbsp/microui/io/SystemInputPump.@134265968
  [0x0800BC80]
  at ej/microui/Pump.@134261696
  [0x0800ABCC]
  at ej/microui/Pump.@134265872
  [0x0800BC24]
  at java/lang/Thread.@134273964
  [0x0800DBC4]
  at java/lang/Thread.@134273784
  [0x0800DB04]
  at java/lang/Thread.@134273892
  [0x0800DB6F]
-----
Java Thread[2]
name="DISPLpmp" prio=5 state=WAITING

java/lang/Thread:
  at java/lang/Object.@134256392
  [0x08009719]
  at ej/microui/FIFOPump.@134259824
  [0x0800A48E]
  at ej/microui/io/DisplayPump.134263016
  [0x0800B0F8]
  at ej/microui/Pump.@134261696
  [0x0800ABCC]
  at ej/microui/Pump.@134265872
  [0x0800BC24]
  at ej/microui/io/DisplayPump.@134262868
  [0x0800B064]
  at java/lang/Thread.@134273964
  [0x0800DBC4]
  at java/lang/Thread.@134273784
  [0x0800DB04]
  at java/lang/Thread.@134273892
  [0x0800DB6F]
=====

```

Example 5.2. MicroJvm Virtual Machine Dump

5.5 Java Language

The MicroJvm Virtual Machine is compatible with the Java language version 7.

5.6 Java Libraries

The MicroJvm Virtual Machine is compliant with the Java Embedded Device Configuraton 1.2 (EDC). This library provides the standard classes for a Java application.

5.6.1 Embedded Device Configuration (EDC)

The Embedded Device Configuration specification defines the minimal standard runtime environment for embedded devices. It defines all the default API packages:

- java.io
- java.lang
- java.lang.annotation

- java.lang.ref
- java.lang.reflect
- java.util

5.6.2 Beyond Profile (B-ON)

B-ON defines a suitable and flexible way to fully control both memory usage and start-up sequences on devices with limited memory resources. It does so within the boundaries of the Java semantic. More precisely, it allows:

- Controlling the initialization sequence in a deterministic way.
- Defining persistent immutable read-only objects (that may be placed into non-volatile memory areas), and do not require copies to be made in ram to be manipulated.
- Defining immortal read-write objects that are always alive.

5.7 Dependencies

The MicroJvm Virtual Machine requires an implementation of its low level APIs to run. Refers to the chapter “Implementation” for more information.

5.8 Installation

The MicroJvm Virtual Machine and its components are already installed in the platform.

5.9 Use

A Java classpath variable named `EDC-1.2` is available, according to the selected Java core library. This Java classpath variable is always required in the build path of a Java project and all others Java libraries depend on it.

Another classpath variable named `BON-1.2` is available. This variable must be added to the build path of the Java application project in order to access the B-ON library.

Some options can be defined for these libraries when the Java application is built. Refer to the chapter “Launch Options” which lists all these options.

6 Native Interface Mechanisms

The MicroJvm Virtual Machine provides two ways to link Java application code with native C code. The two ways are fully complementary and can be used at the same time.

6.1 *Simple Native Interface – Green Thread (SNI-GT)*

6.1.1 Introduction

SNI-GT provides a simple mechanism for implementing native Java methods in C language.

SNI-GT allows to:

- call a C function from a Java method.
- access an Immortal array in a C function (see [B-ON] specification to learn about immortal objects).

SNI-GT does not allow to:

- access or create a Java object in a C function.
- access Java static variables in a C function.
- call Java methods from a C function.

SNI-GT provides some Java APIs to manipulate some data arrays between Java and native (C) world.

6.1.2 Functional Description

SNI-GT defines how to cross the barrier between Java world and native world:

- Call a C function from Java.
- Pass parameters to the C function.
- Return a value from the C world to the Java world.
- Manipulate (read " write) shared memory both in Java and C : the immortal space.

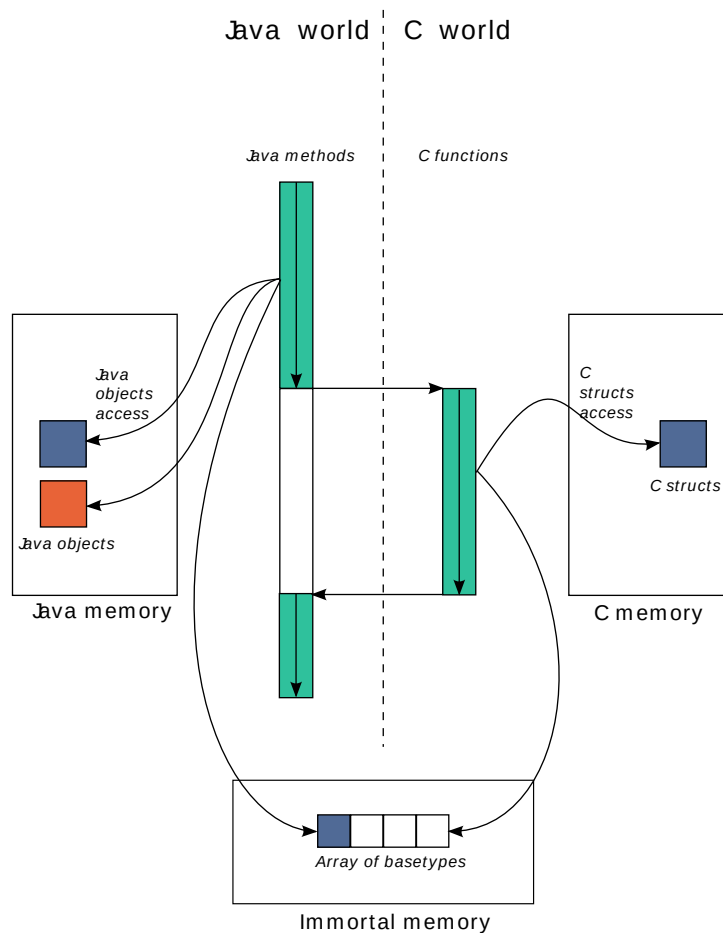


Figure 6.1. SNI Processing

Figure 6.1 illustration shows both Java and C code accesses to shared objects in the immortal space, while also accessing their respective memory.

6.1.3 Synchronization

A call to a native function uses the same RTOS task than the RTOS task used to run all Java green threads. So during this call the MicroJvm Virtual Machine is not able to schedule the others Java threads.

SNI-GT defines C functions that provide controls upon the green threads activities:

- `int32_t SNI_suspendCurrentJavaThread(int64_t timeout)`: suspends the execution of the Java thread that has initiated the current C call. This function does not block the C execution. The suspension is effective only at the end of the native method call (when the C call returns). The green thread is suspended until either a RTOS task calls `SNI_resumeJavaThread` or if the specified amount of milliseconds has elapsed.
- `int32_t SNI_getCurrentJavaThreadID(void)`: permits to retrieve the ID of the current Java thread within the C function (assuming it is a "native Java to C call"). This ID must be given to the `SNI_resumeJavaThread` function in order to resume the green thread execution.
- `int32_t SNI_resumeJavaThread(int32_t id)`: resumes the green thread with given ID. If the thread is not suspended, the resume stays pending.

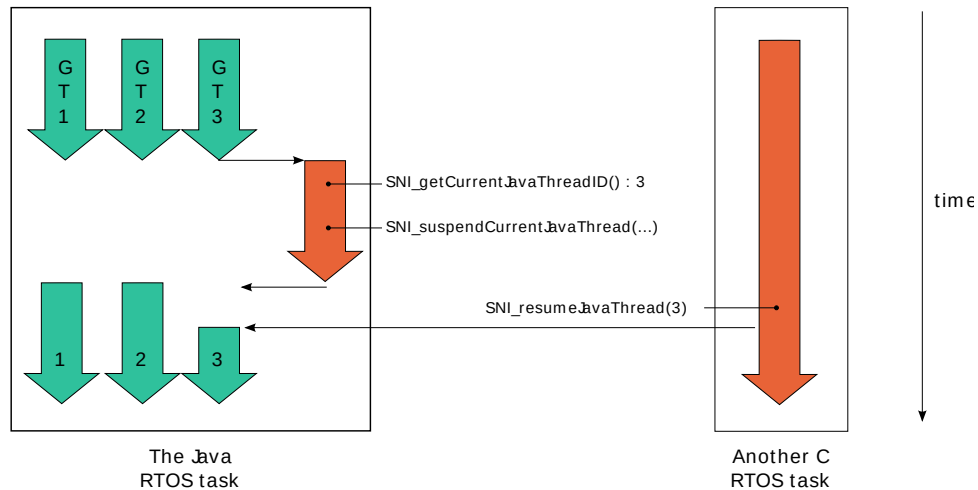


Figure 6.2. Green threads and RTOS task synchronization

Figure 6.2 illustration shows a green thread (GT3) which has called a native method that executes in C. The C code suspends it, after having provisioning its ID (e.g. 3). Another RTOS task may later resume the Java green thread.

6.1.4 Example

```
package example;

import java.io.IOException;

public abstract class Sensor {

    public static final int ERROR = -1;

    public int getValue() throws IOException{
        int sensorID = getSensorID();
        int value = getSensorValue(sensorID);
        if (value == ERROR) {
            throw new IOException("Unsupported sensor");
        }
        return value;
    }

    protected abstract int getSensorID();

    public static native int getSensorValue(int sensorID);
}

class Potentiometer extends Sensor {
    protected int getSensorID() {
        return Constants.POTENTIOMETER_ID; // POTENTIOMETER_ID is a static final
    }
}
```

```
#include <sni.h>
#include <potentiometer.h>

#define SENSOR_ERROR (-1)
#define POTENTIOMETER_ID (3)

jint Java_example_Sensor_getSensorValue(jint sensor_id){
    if (sensor_id == POTENTIOMETER_ID)
    {
        return get_potentiometer_value();
    }
    return SENSOR_ERROR;
}
```

6.1.5 Dependencies

- EDC Java core library (see “Java Libraries”).
- B-ON Java core library (see “Java Libraries”).

6.1.6 Installation

SNI-GT library is a built-in feature of the platform, so there is no additional dependency to call native code from Java. In the platform configuration file, check `Java to C Interface > Simple Native Interface API` to install the additional Java APIs to manipulate the data arrays.

6.1.7 Use

A classpath variable named `SNI-GT-1.1.1` is available, which needs to be added to the build path of the Java application project in order to access the SNI-GT library.

6.2 *Shielded Plug (SP)*

6.2.1 Introduction

The Shielded Plug [SP] provides data segregation with a clear publish-subscribe API. The data sharing between modules uses the concept of shared memory blocks, with introspection. The database is made of blocks: chunks of RAM.

6.2.2 Functional Description

The usage of the Shielded Plug (SP) starts with the definition of a database. The implementation of the SP for the JPF uses an XML file description to describe the database; the syntax follows the one proposed by the SP specification [SP].

Once this database is defined it can be accessed within the Java application or the C application. The SP Java library is accessible from the classpath variable `SP-1.0`. This library contains the classes and methods to read and write data in the database. See also the Java documentation from the MicroEJ workbench resources center ("Javadoc" menu). The C header file `sp.h` available in Java platform source/`MICROJVM/include` folder contains the C functions to access the database.

To embed the SP database in your binary file, the XML file description must be processed by the SP compiler. This compiler generates a binary file (`.o`) that will be linked to the overall application by the linker. It also generates two descriptions of the block ID constants, one in Java and one in C. These constants can be used by either the Java or the C application modules.

A MicroEJ tool is available to launch the SP compiler tool. The tool name is **Shielded Plug Compiler**.

6.2.2.1 Category: Shielded Plug Compiler

Shielded Plug Compiler configuration

Database definition:

C Generation

☐ Generates databases' ID in C header files

Output folder:

C constants' name prefix :

Java Generation

☐ Generates databases' ID in Java interfaces

Output folder:

Output package:

6.2.2.1.1 Group: Shielded Plug Compiler configuration

6.2.2.1.1.1 Option(browse): Database definition

Default value: (empty)

Description:

Choose the database XML definition.

6.2.2.1.2 Group: C Generation

6.2.2.1.2.1 Option(checkbox): Generates databases' ID in C header files

Default value: unchecked

Description:

When checked, databases' ID are generated into C header files.

6.2.2.1.2.2 Option(browse): Output folder

Default value: (empty)

Description:

Folder where C header files are generated.

6.2.2.1.2.3 Option(text): C constants' name prefix

Default value: (empty)

6.2.2.1.3 Group: Java Generation

6.2.2.1.3.1 Option(checkbox): Generates databases' ID in Java interfaces

Default value: unchecked

Description:

When checked, databases' ID are generated into Java interfaces.

6.2.2.1.3.2 Option(browse): Output folder

Default value: (empty)

Description:

Folder where Java interfaces are generated.

6.2.2.1.3.3 Option(text): Output package

Default value: (empty)

6.2.3 Example

Below is an example of using a database SP. The code that publishes the data is written in C, and the code that receives the data is written in Java. The data is transferred using two memory blocks. One is a scalar value, the other is a more complex object representing a two dimensional vector.

6.2.3.1 Database Description

The database is described as follows:

```
<shieldedPlug>
  <database name="Forecast" id="0" immutable="true" version="1.0.0">
    <block id="1" name="TEMP" length="4" maxTasks="1"/>
    <block id="2" name="THERMOSTAT" length="4" maxTasks="1"/>
  </database>
</shieldedPlug>
```

6.2.3.2 Java Code

From the database description we can create a Java interface.

```
public interface Forecast {
    public static final int ID = 0;
    public static final int TEMP = 0;
    public static final int THERMOSTAT = 1;
}
```

Below is the task that reads the published temperature and controls the thermostat.

```

public void run(){
    ShieldedPlug database = ShieldedPlug.getDatabase(Forecast.ID);
    while (isRunning){
        //reading the temperature every 30 seconds
        //and update thermostat status
        try {
            int temp = database.readInt(Forecast.TEMP);
            print(temp);
            //update the thermostat status
            database.writeInt(Forecast.THERMOSTAT,temp>tempLimit ? 0 : 1);
        }
        catch(EmptyBlockException e){
            print("Temperature not available");
        }
        sleep(30000);
    }
}

```

6.2.3.3 C Code

C header that declares the constants defined in the XML description of the database.

```

#define Forecast_ID 0
#define Forecast_TEMP 0
#define Forecast_THERMOSTAT 1

```

Publication of temperature and thermostat controller task.

```

void temperaturePublication(){
    ShieldedPlug database = SP_getDatabase(Forecast_ID);
    int32_t temp = temperature();
    SP_write(database, Forecast_TEMP, &temp);
}

void thermostatTask(){
    int32_t thermostatOrder;
    ShieldedPlug database = SP_getDatabase(Forecast_ID);
    while(1){
        SP_waitFor(database, Forecast_THERMOSTAT);
        SP_read(database, Forecast_THERMOSTAT, &thermostatOrder);
        if(thermostatOrder == 0) {
            thermostatOFF();
        }
        else {
            thermostatON();
        }
    }
}

```

6.2.4 LLSP: Low Level SP API

The implementation of the SP for the JPF assumes some support from the underlying RTOS. It is mainly related to provide some synchronization when reading / writing into Shielded Plug blocks.

- LLSP_IMPL_syncWriteBlockEnter and LLSP_IMPL_syncWriteBlockExit are used as a semaphore by RTOS tasks. When a task wants to write to a block, it "locks" this block until it has finished to write in it.
- LLSP_IMPL_syncReadBlockEnter and LLSP_IMPL_syncReadBlockExit are used as a semaphore by RTOS tasks. When a task wants to read a block, it "locks" this block until it is ready to release it.

The [SP] specification provides a mechanism to force a task to wait until new data has been provided to a block. The implementation relies on functions `LLSP_IMPL_wait` and `LLSP_IMPL_wakeup` to block the current task and to reschedule it.

The file `LLSP_impl.h` that comes with the JPF defines the API to be implemented.

6.2.5 Dependencies

- EDC Java core library (see “Java Libraries”).
- B-ON Java core library (see “Java Libraries”).

6.2.6 Installation

SP library and its relative tools is an optional feature of the platform. In the platform configuration file, check `Java to C Interface > Shielded Plug` to install the it.

6.2.7 Use

A classpath variable named `SP- 1.0` is available, which needs to be added to the build path of the Java application project in order to access the SP library.

There is a configuration option in the MicroEJ launchers to specify the SP database structure definition file. Please refer to Section 12 to see the location of that option.

6.3 MicroEJ Java H

6.3.1 Introduction

This MicroEJ tool is useful to create the skeleton of a C file where some Java native implementation functions will be written later. This tool prevents to miss some `#include` and ensure the functions signatures are correct.

6.3.2 Functional Description

MicroEJ Java H tool takes in input one or several Java class files (`*.class`) from directories and / or JAR files. It looks for Java native methods declared in these class files and generate skeleton(s) of C file(s).

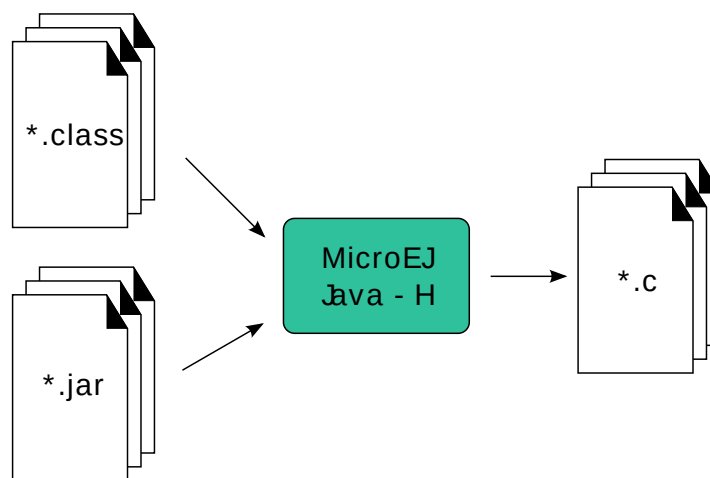


Figure 6.3. MicroEJ Java H Process

6.3.3 Dependencies

No dependency is required.

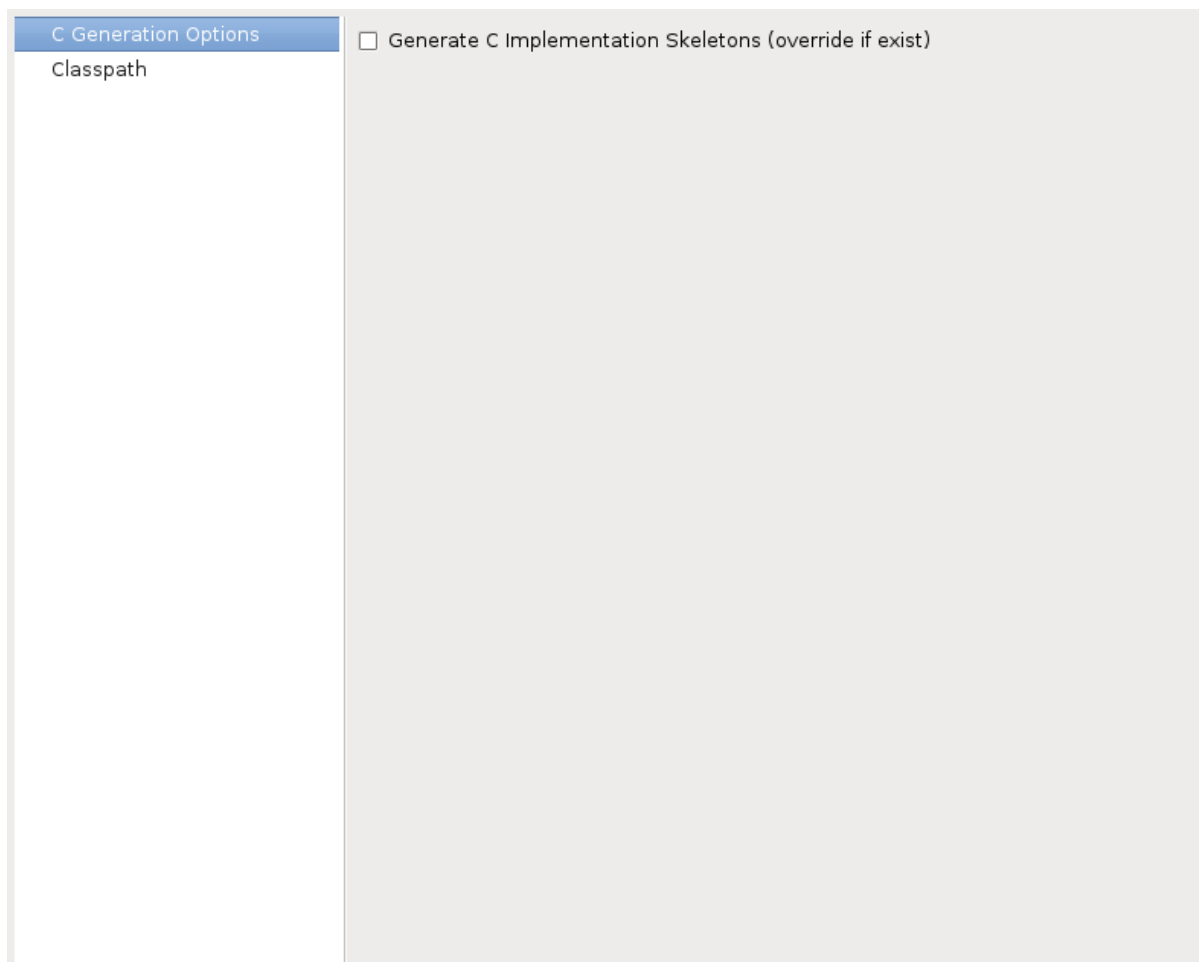
6.3.4 Installation

This is an additional tool. In the platform configuration file, check `Java to C Interface > MicroEJ Java H` to install the it.

6.3.5 Use

This chapter explains the MicroEJ tool options.

6.3.5.1 Category: C Generation Options



Description: Define the C Generation options

6.3.5.1.1 Option(checkbox): Generate C Implementation Skeletons (override if exist)

Default value: unchecked

6.3.5.2 Category: Classpath

C Generation Options
Classpath

Define the classpath to look for native declarations:

Add Jar...
Add Directory...
Remove
Up
Down

Description: Define the classpath to look for native declarations

6.3.5.2.1 Option(list): Define the classpath to look for native declarations

Default value: (empty)

7 Embedded Communications

MicroEJ provides some Java libraries to instantiate some communications with external devices. Each communication way has got its own Java library. A global library called ECOM provides an abstract communication stream support (communication framework only).

7.1 ECOM

7.1.1 Introduction

The Embedded COMMunication Java library (ECOM) is a generic communication library which abstract communication stream support (communication framework only). It allows to open and use some streams on communication devices such as a COMM port.

This library does not provide APIs to manipulate some specific options for each communication ways but it provides some generic APIs which abstract the communication way. After the opening step Java application can use every communications (COMM, USB etc.) as generic communication in order to change easily communication way if needed.

7.1.2 Functional Description

Figure 7.1 shows the overall process to open a connection on a hardware device.

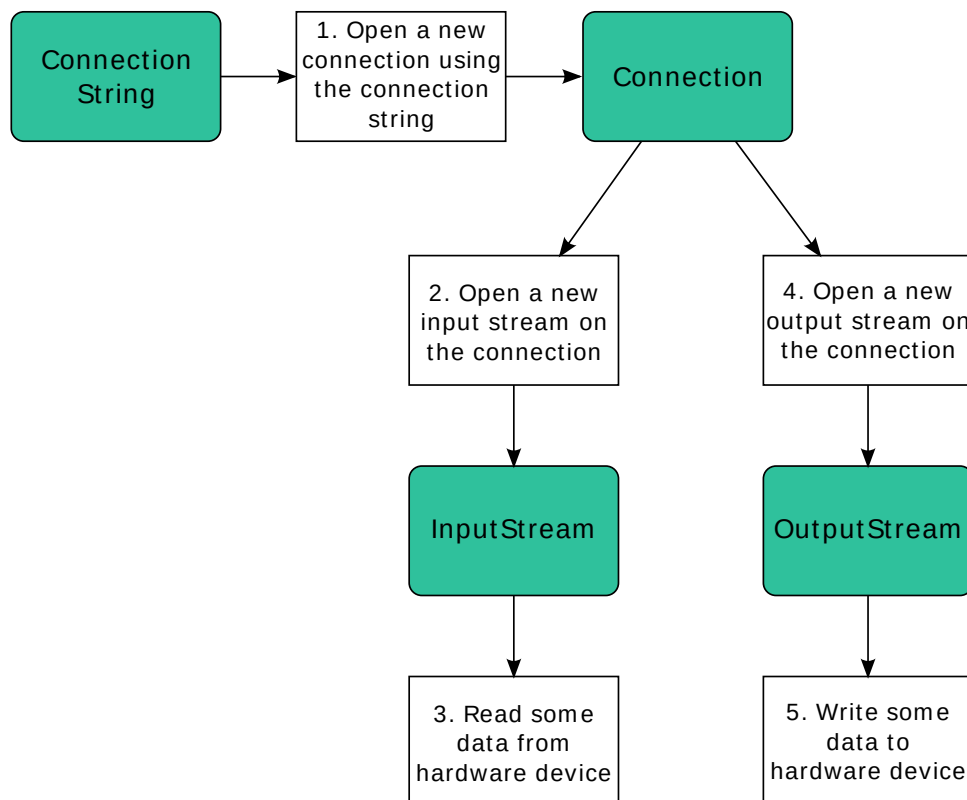


Figure 7.1. ECOM Flow

1. Step 1 consists in opening a Java connection on an hardware device. The connection kind and its configuration is fixed by the parameter String `connectionString` of the method `Connection.open`.
2. Step 2 consists in opening an `InputStream` on the connection. This stream allows to the Java application to access the "RX" feature of the hardware device.
3. Step 3 consists in using the `InputStream` APIs to receive in Java all hardware device data.
4. Step 4 consists in opening an `OutputStream` on the connection. This stream allows to the Java application to access the "TX" feature of the hardware device.

5. Step 5 consists in using the `OutputStream` APIs to transmit some data from Java to hardware device.

Note that the steps 2 and 4 may be parallelized and do not depend each other.

7.1.3 Dependencies

- EDC Java core library (see “Java Libraries”).

7.1.4 Installation

ECOM Java library is an additional library. In the platform configuration file, check `ECOM > ECOM` to install it.

7.1.5 Use

A classpath variable named `ECOM-1.0` is available. This Java library is always required when developing a Java application which communicates with some external devices. It is automatically embedded as soon as a sub communication library is added in the Java classpath.

This library provides a set of options. Refer to the chapter “Launch Options” which lists all options.

7.2 ECOM Comm

7.2.1 Introduction

The ECOM Comm Java library provides support for serial (UART) communication. ECOM Comm extends ECOM to allow stream communication via serial communication ports (typically UARTs). In the Java application, the connection is established using the `Connector.open()` method. The returned connection is a `StreamConnection`, and the input and output streams can be used for full duplex communication.

The use of ECOM Comm in a custom platform requires the implementation of an UART driver. There are two different modes of communication:

- In Buffered mode, ECOM Comm manages software FIFO buffers for transmission and reception of data. The driver copies data between the buffers and the UART device.
- In Custom mode, the buffering of characters is not managed by ECOM Comm. The driver has to manage its own buffers to make sure no data is lost in serial communications because of buffer overruns.

7.2.2 Functional Description

The ECOM Comm process respects the ECOM process. Please refer to the illustration “ECOM Flow”.

7.2.3 Component architecture

The ECOM Comm C module relies on a native driver to perform actual communication on the serial ports. Each port can be bound to a different driver implementation, but most of the time, it is possible to use the same implementation (i.e. same code) for multiple ports. Exceptions are the use of different hardware UART types, or the need for different behaviors.

Four C header files are provided:

- `LLCOMM_BUFFERED_CONNECTION_impl.h`
Defines the set of functions that the driver must implement to provide a Buffered connection
- `LLCOMM_BUFFERED_CONNECTION.h`
Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Buffered connection

- `LLCOMM_CUSTOM_CONNECTION_impl.h`
Defines the set of functions that the driver must implement to provide a Custom connection
- `LLCOMM_CUSTOM_CONNECTION.h`
Defines the set of functions provided by ECOM Comm that can be called by the driver (or other C code) when using a Custom connection

The ECOM Comm drivers are implemented using standard LLAPI features. The diagram below shows an example of the objects (both Java and C) that exist to support a Buffered connection.

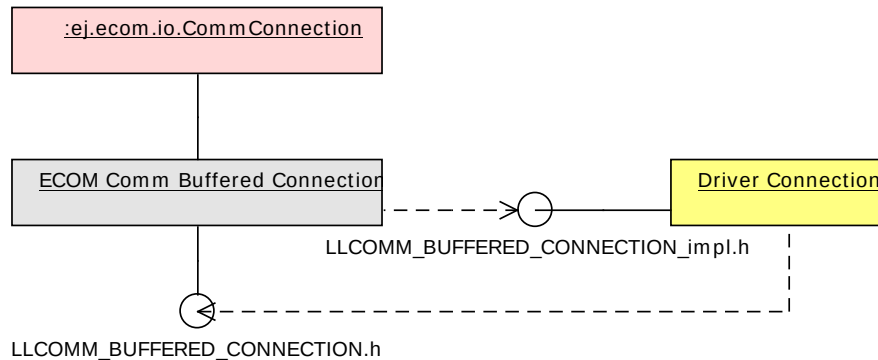


Figure 7.2. ECOM Comm components

The connection is implemented with three objects ¹ :

- The Java object used by the application; an instance of `ej.ecom.io.CommConnection`
- The connection object within the ECOM Comm C module
- The connection object within the driver

Each driver implementation provides one or more connections. Each connection typically corresponds to a physical UART.

7.2.4 Logical port mapping

Each serial port available for use in ECOM Comm can be identified in two ways:

- A logical port number. This identifier is specific to the application, and should be used to identify the data stream that the port will carry (for example, "debug traces" or "GPS data").
- A physical port number. This is specific to the hardware, and identifies which UART device and I/O pins will be used for communication ².

The mapping from logical port numbers to physical ports is done in the application launch configuration. This way, the application can refer only to the logical port number, and the data stream can be directed to the matching I/O port on different versions of the hardware.

Ultimately, the logical port number is only visible to the application. The physical identifier will be sent to the driver.

¹This is a conceptual description to aid understanding - the reality is somewhat different, although that is largely invisible to the implementor of the driver.

²Some drivers may reuse the same UART device for different ECOM ports with a hardware multiplexer. Drivers can even treat the physical port number as a logical id and map the ids to various I/O channels.

7.2.5 Java API

Opening a connection is done using `ej.ecom.io.Connector.open(String name)`. The connection string (the name parameter) must start with "comm:", followed by the Comm port identifier ("com" + a logical port number), and a semicolon-separated list of options. Options are the baudrate, the parity, the number of bits per character, and the number of stop bits:

- baudrate=n
- bitsperchar=n where n is in the range 5 to 9
- stopbits=n where n is 1, 2, or 1.5
- parity=x where x is odd, even or none

All of these are optional. Illegal or unrecognized parameters cause an `IllegalArgumentException`.

7.2.6 Driver API

The ECOM Comm Low Level API is designed to allow multiple implementations (e.g. drivers that support different UART hardware) and connection instances (see Low Level API Pattern in [CMREF]). Each ECOM Comm driver defines a data structure that holds information about a connection, and functions take an instance of this data structure as the first parameter.

The name of the implementation must be set at the top of the driver C file, for example³:

```
#define LLCOMM_BUFFERED_CONNECTION_IMPL MY_LLCOMM
```

This defines the name of this implementation of the `LLCOMM_BUFFERED_CONNECTION_IMPL` interface to be `MY_LLCOMM`.

The data structure managed by the implementation must look like this:

```
typedef struct MY_LLCOMM{
    struct LLCOMM_BUFFERED_CONNECTION header;
    // extra data goes here
} MY_LLCOMM;

void MY_LLCOMM_new(MY_LLCOMM* env);
```

In this example the structure contains only the default data, in the header field. Note that the header must be the first field in the structure. The name of this structure must be the same as the implementation name (`MY_LLCOMM` in this example).

The driver must also declare the "new" function used to initialize connection instances. The name of this function must be the implementation name with `_new` appended, and it takes as its sole argument a pointer to an instance of the connection data structure, as shown above.

The driver needs to implement the functions specified in the `LLCOMM_BUFFERED_CONNECTION_impl.h` (or `LLCOMM_CUSTOM_CONNECTION_impl.h`) file.

The ECOM Comm C module needs to know, when the Java application is built, the name of the implementation. This mapping is defined in an Board Support Package configuration file (see "XML File"). In this example the XML file would contain the line:

```
<nativeImplementation
    name="MY_LLCOMM"
    nativeName="LLCOMM_BUFFERED_CONNECTION_IMPL"
/>
```

³The following examples use Buffered connections, but Custom connections follow the same pattern.

where `nativeName` is the name of the interface, and `name` is the name of the implementation.

This `ilink` file must reside in the `source/lib` folder of your work-in-progress JPF project when the Java application is built.

The driver defines the set of connections it provides by making an entry for each connection in the `COMM_CONNECTIONS_TABLE`. The entry is a pointer to a function that will initialize the data structure that represents the connection object and return it. The last entry in the table must have the value 0. For example, if the `MY_LLCOMM` driver implementation has two connection instances then the table would look like this:

```
void* COMM_CONNECTIONS_TABLE[] = {  
    /* pointer to function that creates the first connection */,  
    /* pointer to function that creates the second connection */,  
    0  
};
```

When opening a port from the Java application, each connection declared in the table will be asked (using the `canOpen` method) if it matches the requested physical port identifier. The first connection that returns `true` is used.

The life of a connection starts with the call to `canOpen()`. If the driver indicates that it can open the connection, the connection will be initialized, configured and enabled. Notifications and interrupts are then used to keep the stream of data going. When the connection is closed by the application, interrupts are disabled and the driver will not receive any more notifications. It is important to remember that the transmit and receive sides of the connection are separate Java stream objects, thus, they may have a different life cycle and one side may be closed long before the other.

7.2.6.1 The Buffered Comm stream

In Buffered mode, two buffers are allocated by the driver for sending and receiving data. The `ECOM Comm C` module will fill the transmit buffer, and get bytes from the receive buffer. There is no flow control.

When the transmit buffer is full, an attempt to write more bytes from the Java application will block the Java thread trying to write, until some characters are sent on the serial line and space in the buffer is available again.

When the receive buffer is full, characters coming from the serial line will be discarded. The driver must allocate a buffer big enough to avoid this, according to the UART baudrate, the expected amount of data to receive, and the speed at which the application can handle it.

The Buffered C module manages the characters sent by the application and stores them in the transmit buffer. On notification of available space in the hardware transmit buffer, it handles removing characters from this buffer and putting them in the hardware buffer. On the other side, the driver notifies the C module of data availability, and the C module will get the incoming character. This character is added to the receive buffer and stays there until the application reads it.

The driver should take care of the following:

- Setting up interrupt handlers on reception of a character, and availability of space in the transmit buffer. The C module may mask these interrupts when it needs exclusive access to the buffers. If no interrupt is available from the hardware or underlying software layers, it may be faked using a polling thread that will notify the C module.
- Initialization of the I/O pins, clocks, and other things needed to get the UART working.
- Configuration of the UART baudrate, character size, flow control and stop bits according to the settings given by the C module.
- Allocation of memory for the transmit and receive buffers.

- Getting the state of the hardware: is it running, is there space left in the TX and RX hardware buffers, is it busy sending or receiving bytes?

The driver is notified on the following events:

- Opening and closing a connection: the driver must activate the UART and enable interrupts for it.
- A new byte is waiting in the transmit buffer and should be copied immediately to the hardware transmit unit. The C module makes sure the transmit unit is not busy before sending the notification, so it is not needed to check for that again.

The driver must notify the C module on the following events:

- Data has arrived that should be added to the receive buffer (using the `LLCOMM_BUFFERED_CONNECTION_dataReceived` function)
- Space available in the transmit buffer (using the `LLCOMM_BUFFERED_CONNECTION_transmitBufferReady` function)

7.2.6.2 The Custom Comm stream

In custom mode, the ECOM Comm C module will not do any buffering. Read and write requests from the application are immediately forwarded to the driver.

Since there is no buffer on the C module side when using this mode, the driver has to define a strategy to store received bytes that were not handed to the C module yet. This could be a fixed or variable size FIFO, the older received but unread bytes may be dropped, or a more complex priority arbitration could be set up. On the transmit side, if the driver does not do any buffering, the Java thread waiting to send something will be blocked and wait for the UART to send all the data.

In Custom mode flow control (eg. RTS/CTS or XON/XOFF) can be used to notify the device connected to the serial line and so avoid losing characters.

7.2.7 XML File

The Java platform has to know the available number of Comm ports the Java application will be able to use. Each Comm port is identified by its port number and by an optional nickname (this nickname will be visible in the MicroEJ launcher options, see “Launch Options”).

A XML file is so required to configure the Java platform. The name of this file must be `ecom-comm.xml`. It has to be stored in the module configuration folder (see “Installation”).

This file must start with the node `<ecom>` and the sub node `<comms>`. It can contain several time this kind of line: `<comm port="A_COMM_PORT_NUMBER" nickname name="A_NICKNAME"/>` where:

- `A_COMM_PORT_NUMBER` refers the Comm port the Java platform user will be able to use. This number is a physical number or a logical number known by the Comm driver (see “Logical port mapping”)
- `A_NICKNAME` is optional. It allows to fix a printable name of the Comm port.

Example:

```
<ecom>
  <comms>
    <comm port="2"/>
    <comm port="3" nickname="DB9"/>
    <comm port="5"/>
  </comms>
</ecom>
```

Example 7.1. ecom-comm.xml

First Comm port holds the port 2, second "3" and last "5". Only the second Comm port holds a nickname "DB9".

7.2.8 ECOM Comm Mock

In the simulation environment, no driver is required. The ECOM Comm mock handles communication for all the serial ports and can redirect each port to one of the following:

- An actual serial port on the host computer: any serial port identified by your operating system can be used. The baudrate and flow control settings are forwarded to the actual port.
- A TCP socket. You can connect to a socket on the local machine and use netcat or telnet to see the output, or you can forward the data to a remote device.
- Files. You can redirect the input and output each to a different file. This is useful for sending precomputed data and looking at the output later on for offline analysis.

When using the socket and file modes, there is no simulation of an UART baudrate or flow control. On a file, data will always be available for reading and will be written without any delay. On a socket, you can reach the maximal speed allowed by the network interface.

7.2.9 Dependencies

- CLDC or EDC Java core library (see “Java Libraries”).
- ECOM (see “ECOM”).

7.2.10 Installation

ECOM-Comm Java library is an additional library. In the platform configuration file, check `ECOM > ECOM COMM` to install it. When checked, the xml file `ecom-comm > ecom-comm.xml` is required during platform creation to configure the module (see “XML File”).

7.2.11 Use

A classpath variable named `ECOM-COMM- 1.0` is available. This Java library is always required when developing a Java application which communicates with some external devices using the serial communication mode.

This library provides a set of options. Refer to the chapter “Launch Options” which lists all options.

8 Additional Java Libraries

MicroEJ provides some additional modules which can be used to customize the platform or its environment during its creation.

8.1 Native Language Support (NLS)

8.1.1 Introduction

The NLS library facilitates internationalization. It provides support to manipulate messages and translate them in different languages.

Each message for which there will be an alternative translation is given a logical name (the *message name*), and the set of messages is itself identified by a name, called the *header*.

Each language for which messages translations exist is identified by a string called the *locale*. The format of the locale string is not restricted but by convention it is the concatenation of a language code and a country code:

- The language code is a lower-case, two-letter code as defined by ISO-639.
- The country code is an upper-case, two-letter code as defined by ISO-3166.

Therefore, the required message string is obtained by specifying the *header*, the *locale* and the *message name*.

The NLS data is defined using a combination of Java interfaces and text files. The message strings are pre-processed into immutable objects, available to the NLS library at runtime.

8.1.2 Functional Description

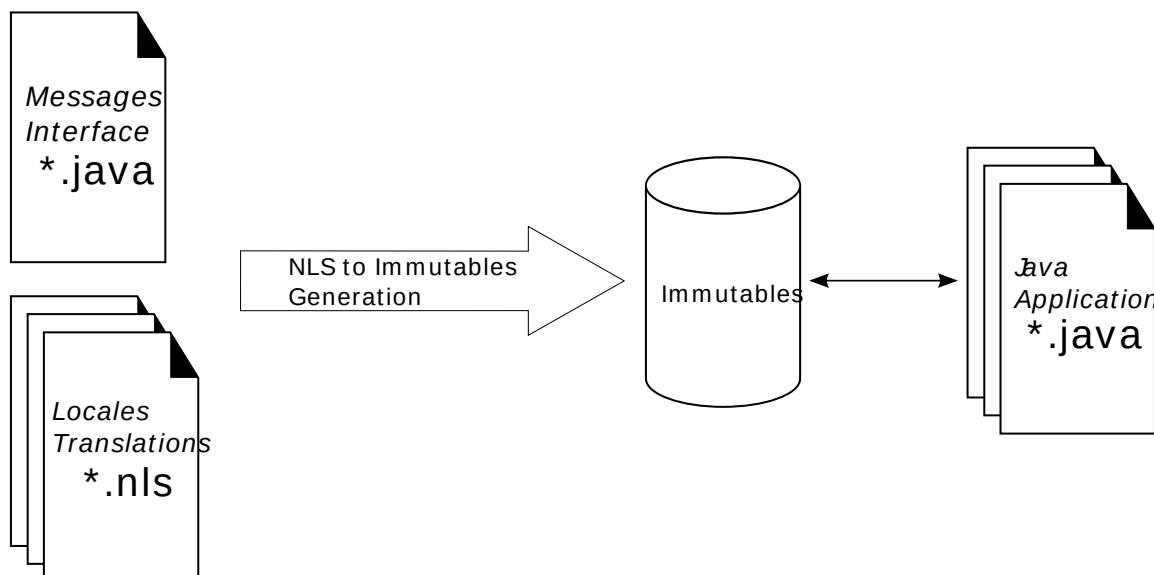


Figure 8.1. Native Language Support Process

The header and message names are specified by a Java interface. The name of the interface is the *header*. It contains a constant (`public static final int`) for each message. The name of the field is the *message name*. The values of the fields must form a contiguous range of integers starting at 1. Here is an example:

```
package com.is2t.appnotes.nls;

public interface HelloWorld {

    public static final int HELLO_WORLD = 1;

}
```

The application can define multiple headers, each specified by a separate Java interface.

For each locale, a properties file is defined that will translate all messages and define the language printable name (`DISPLAY_NAME`). Beware that:

- the file name matches `[header]_[locale].nls` .
- the messages keys match (case sensitive) the constants defined in the interface.

English NLS file `helloworld_en_US.nls` :

```
DISPLAY_NAME=English
HELLO_WORLD=Hello world!
```

To be available at runtime, the list of messages must be defined in a file containing the list of the fully-qualified names of the messages set interfaces. For example:

```
com.is2t.appnotes.nls.HelloWorld
```

Then, this file must be referenced in the launcher. See “Launch Options” for more information. The messages will pre-processed into immutable files.

The usage of these messages (converted in immutables) is allowed by creating a `BasicImmutableNLS` instance passing the lower cased header name as argument:

```
NLS nls = new BasicImmutableNLS("helloworld");
```

The messages can be referenced by using `NLS.getMessage(int)` method passing a message constant as argument:

```
String message = nls.getMessage(HelloWorld.HELLO_WORLD);
```

The current locale can be changed using `NLS.setCurrentLocale(String)` method passing the string representing the locale as argument:

```
nls.setCurrentLocale("en_US");
```

The available locales list can be retrieved with `NLS.getAvailableLocales()` method:

```
String[] locales = nls.getAvailableLocales();
```

8.1.3 Dependencies

- EDC Java core library (see “Java Libraries”).
- B-ON Java core library (see “Java Libraries”).

8.1.4 Installation

NLS Java library is an additional library. In the platform configuration file, check `Miscellaneous > Native Language Support` to install it.

8.1.5 Use

A classpath variable named `NLS- 1.0.1` is available.

This library provides a set of options. Refer to the chapter “Launch Options” which lists all options.

8.2 Logging

8.2.1 Introduction

Logging library helps managing log messages within an application. It allows to finely activate or deactivate message output thanks to levels of importance.

8.2.2 Functional Description

You can create different Loggers for different purposes hence splitting messages by theme. A logger offers methods to log a message with a level of importance. The possible levels are predefined by the class `Level`.

Each logger has an associated level and a list of `Handlers`. When you log a message, the logger will compare its own level to the message level. If the message level is lower, nothing happens. If it is higher, then the logger create a new `LogRecord` with the message and its level and forwards it to the registered handlers. Handlers are responsible for outputting the message. They can either write it to the standard output, to a file, etc. They can add special message formatting.

Loggers are efficient to provide the appropriate level of information at a particular moment, in a particular situation. During development, programmers may want to get as much information as possible. This is done by setting a low level to the relevant loggers. Once the application has been deployed, levels may be increased so as to output only important messages.

A logger level and its handlers can be changed dynamically. Setting a logger level to `Level.OFF` before compilation will produce an application that will output nothing. Nevertheless, the code is embedded so output can be turned on during execution.

8.2.3 Dependencies

- EDC Java core library (see “Java Libraries”).

8.2.4 Installation

Logging Java library is an additional library. In the platform configuration file, check `Miscellaneous > Logging Embedded` to install it.

8.2.5 Use

A classpath variable named `LOGGING-EMBEDDED 1.0` is available.

This library does not provide any option.

8.3 Components

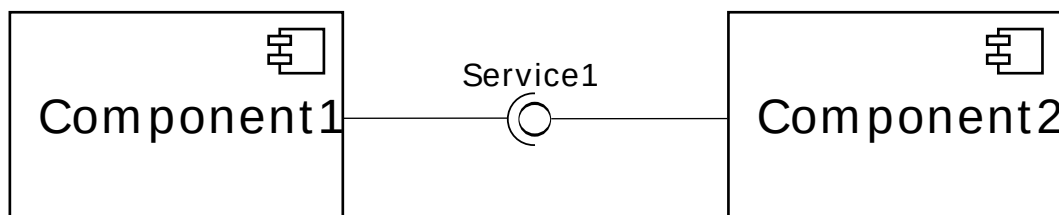
8.3.1 Introduction

Components library helps complex applications development by following component-based development (CBD) and/or service-oriented architecture (SOA) concepts.

The final application is composed of several bundles (or modules) that provide one or several services. The bundles only depends on API between each other (loose coupling). This way:

- each bundle can be developed, modified or replaced easily without impacting the integrity of the whole application.
- Bundles (and associated services) can be reused from an application to another.
- An application can easily be adapted on a target or another just by replacing one or several bundles (for example using a different communication layer).

8.3.2 Functional Description



Component1 requires Service1. An implementation of this service is provided by Component2.

Figure 8.2. Service Oriented Architecture

A service is a Java interface. A service implementation is the class (or set of classes) implementing a service interface.

A bundle is a services implementations provider that implements `BundleActivator`. It defines 4 phases (its life-cycle):

- **Initialization.** The bundle initializes and declares its services implementations. It is done by calling `Registry.register(Class, Object)` method passing the interface class and a implementation instance. This method shall initialize just a few things (ideally just creating the implementation instance), the main part of the initialization shall be done in the starting phase.
- **Link.** The bundle retrieves services implementations and linked it to its services implementations. The method `Registry.getService(Class)` returns the service implementation (if exists) registered for the given service.
- **Start.** The bundle starts its services implementations. In this phase, the services may starts threads or tasks, allocates structures or resources, etc.
- **Stop.** The bundle stops its services implementations. It must stop and clean potential threads and resources created in the precedent phases (all the bundles relative instances must be eligible to garbage collection after this phase).

Components library central feature is the registry of services. A registry is a bundle that includes other bundles. It concentrates services registration to simplify connection between services. A service can be queried from the registry using `Registry.getService()` method.

The registry needs to be started up by calling sequentially

```
registry.initialize(parameters);
registry.link(parameters);
registry.start(parameters);
```

or just

```
RegistryFactory.startupRegistry(registry, parameters);
```

All registry bundles will also be started up following each phase. This ensure that all bundles (and provided services implementations) are fully initialized before linking and are fully linked and functional before starting.

A unique instance of the registry can be used with `RegistryFactory.getRegistry()` method. This instance can be specified by setting `ej.components.Registry` property to the fully qualified name of the registry implementation (for example `ej.components.util.FileRegistry` for the file-based implementation).

An implementation of the registry based on a file description is available:

- create a file containing the list of bundles to activate with their parameters. One declaration by line in the form: `bundle.fully.qualified.name:parameters` . The parameters can be empty. Each bundle can define its own parameters format.
- Startup the file registry with the path to the file as parameter. For example:

```
RegistryFactory.startupRegistry(registry, "/components/mycomponents.properties")
```

It is also possible to define its own implementation.

8.3.3 Dependencies

- EDC Java core library (see “Java Libraries”).

8.3.4 Installation

Components Java library is an additional library. In the platform configuration file, check `Miscellaneous > Components` to install it.

8.3.5 Use

A classpath variable named `EJ.COMPONENT- 1.0` is available.

This library does not provide any option.

9 Development Tools

MicroEJ provides several development tools to help to develop and debug the Java application. Some tools are common for the Embedded platform (EmbJPF) and for the Simulator (SimJPF), some others are only for one of both.

9.1 Memory Map Analyzer

9.1.1 Introduction

When a Java application is linked with the MicroEJ workbench, a Memory MAP file is generated. The Memory Map Analyzer (MMA) is an Eclipse plug-in made for exploring the map file. It displays the memory consumption of different features in the RAM and ROM.

9.1.2 Functional Description

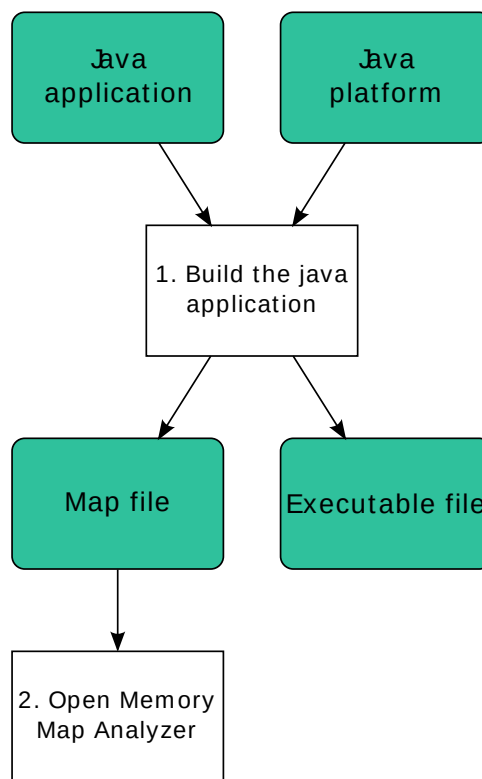


Figure 9.1. Memory Map Analyzer Process

In addition with the executable file, the Java platform generates a map file. Double click on this file to open the Memory Map Analyzer.

9.1.3 Dependencies

No dependency is required.

9.1.4 Installation

This tool is a built-in platform tool. There is nothing else to install in the platform to be able to use it.

9.1.5 Use

The map file is available in the Java application project output directory.

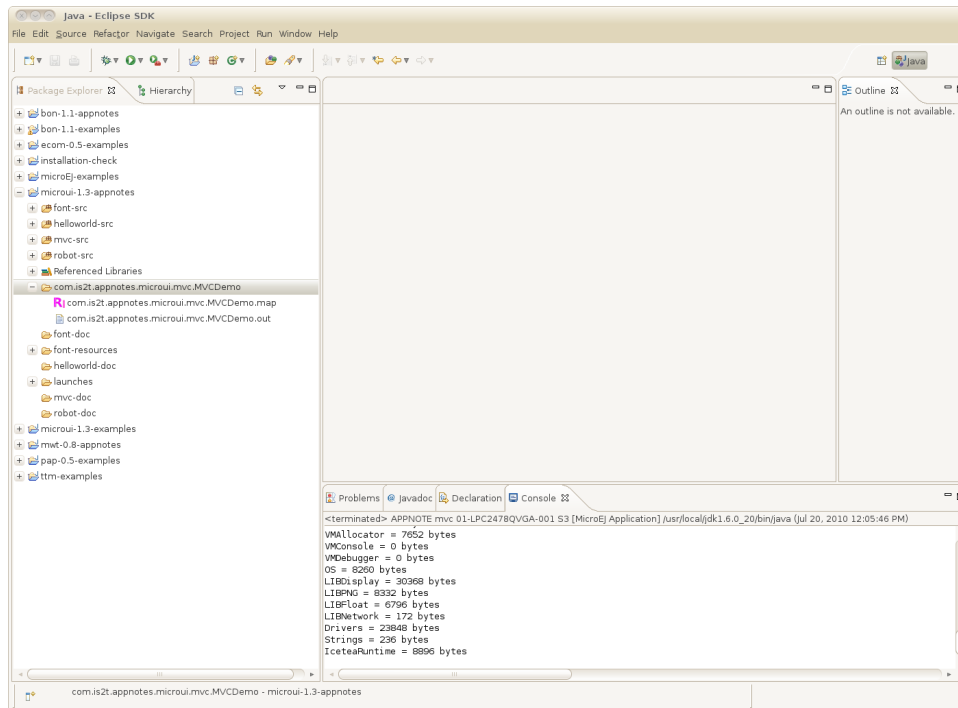


Figure 9.2. Retrieve Map File

Select a (or several) items to show the memory used by this (these) item(s) on the right. Select "All" item to show the memory used by all items. This special item performs the same action as selected all items in the list.

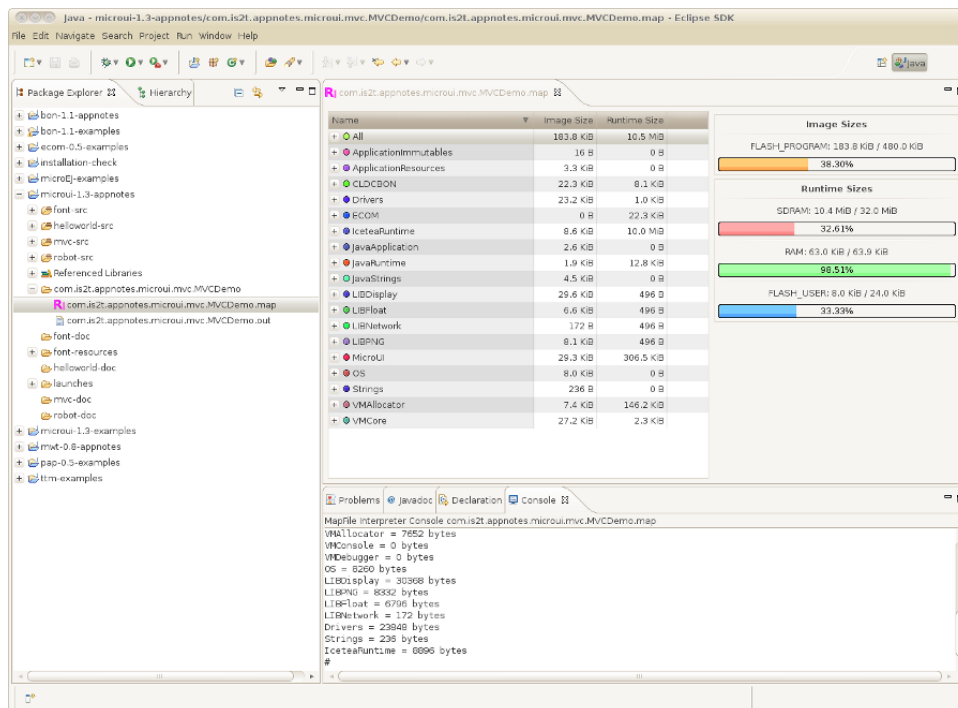


Figure 9.3. Consult Full Memory

Select an item in the list and expand it to see all symbols used by the item. This view is useful to understand why a symbol is embedded.

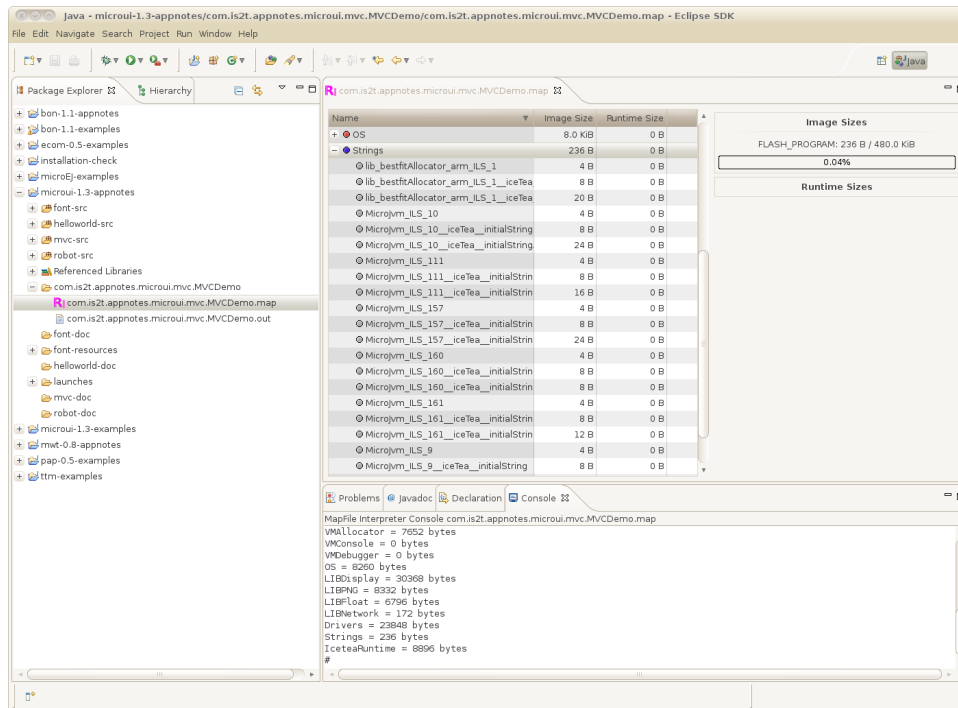


Figure 9.4. Detailed View

9.2 Stack Trace Descriptor

9.2.1 Introduction

Stack Trace Descriptor is a MicroEJ tool which decodes the Java stack traces. When a Java exception occurs the MicroJVM Virtual Machine prints the stack trace on the standard output `System.out`. The classes names and methods names obtained are encoded with a MicroEJ internal format. This internal format prevents to embed all classes names and methods names in the flash in order to save some memory spaces. The Stack Trace Descriptor tool allows to decode the stack traces by replacing the internal classes names and methods names with their real names. It also retrieves the line number in the Java application.

9.2.2 Functional Description

The Stack Trace Descriptor reads the debug info from the fully linked ELF file (ELF file containing the MicroJVM Virtual Machine, the others libraries, the BSP, the OS and the compiled Java application.). It prints the decoded stack trace.

9.2.3 Dependencies

No dependency is required.

9.2.4 Installation

This tool is a built-in platform tool. There is nothing else to install in the platform to be able to use it.

9.2.5 Use

This chapter explains the MicroEJ tool options.

9.2.5.1 Category: MicroJvm Stack Trace Decrypter

The screenshot shows the 'MicroJvm Stack Trace Decrypt' dialog box. It features a title bar and a left sidebar. The main content area is divided into two sections. The top section, 'Java Application Definition', includes an 'Executable file:' text box and a 'Browse...' button. The bottom section, '"Trace port" interface for Eclipse', contains a 'Connection type:' dropdown menu set to 'Console', a 'Port:' text box with '/dev/ttyS0', a 'Baudrate:' dropdown menu set to '115200', another 'Port:' text box with '5555', an 'Address:' text box, and a 'Microjvm stack trace file:' text box with a 'Browse...' button.

9.2.5.1.1 Group: Java Application Definition

9.2.5.1.1.1 Option(browse): Executable file

Default value: (empty)

Description:

Specify the full path of a full linked elf file.

9.2.5.1.2 Group: "Trace port" interface for Eclipse

Description:

This group describes the hardware link between the board and the PC.

9.2.5.1.2.1 Option(combo): Connection type

Default value: Console

Available values:

Uart (COM)

Socket

File

Console

Description:

Specify the connection type between the Board and PC.

9.2.5.1.2.2 Option(text): Port

Default value: /dev/ttyS0

Description:

Format: port name

Specifies the PC COM port:

Windows - COM1, COM2, ..., COMn

Linux - /dev/ttyS0, /dev/ttyS1, ..., /dev/ttySn

9.2.5.1.2.3 Option(combo): Baudrate

Default value: 115200

Available values:

9600

38400

57600

115200

Description:

Defines the COM baudrate for PC-Board communication (only used when Run MicroJVM Proxy Launch configuration is set)

9.2.5.1.2.4 Option(text): Port

Default value: 5555

Description:

IP port

9.2.5.1.2.5 Option(text): Address

Default value: (empty)

Description:

IP address, on the form A.B.C.D.

9.2.5.1.2.6 Option(browse): MicroJVM stack trace file

Default value: (empty)

9.3 Code Coverage Analyzer

9.3.1 Introduction

The SimJPF features an option to output .cc (Code Coverage) files representing the use rate of functions of an application. It traces how the opcodes are really executed.

9.3.2 Functional Description

Refer to Section 12 , SimJPF MicroEJ launcher options to know how to activate SimJPF Code Coverage option. The Code Coverage Analyzer scans the output .cc files and outputs an HTML report to ease the analysis of methods coverage. The HTML report is available in a folder named htmlReport in the same folder as .cc files..

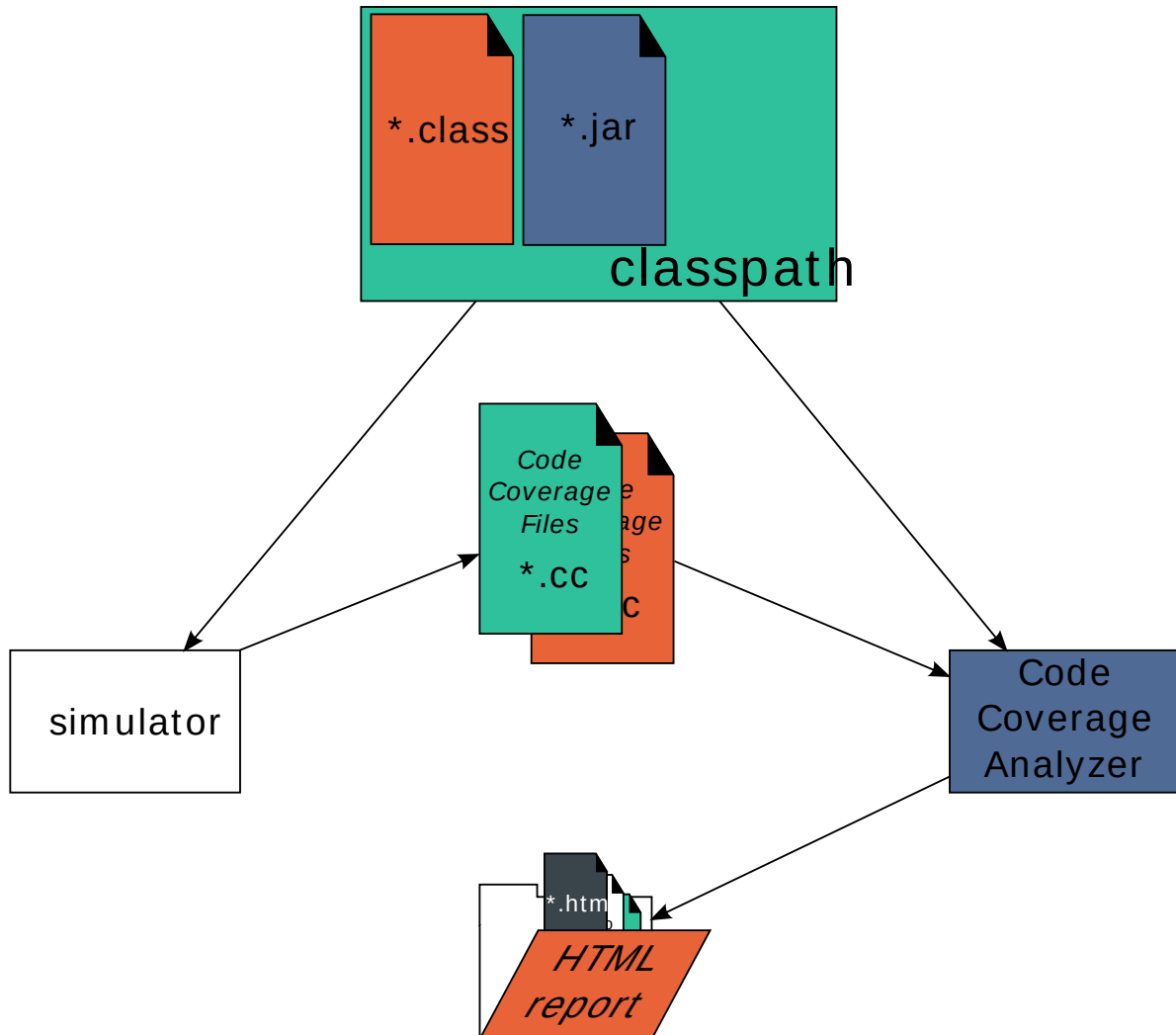


Figure 9.5. Code Coverage Analyzer Process

9.3.3 Dependencies

To work the Code Coverage Analyzer should be input the .cc files. The .cc files relay the classpath used during the execution of the simulator to the Code Coverage Analyzer, therefore the classpath is considered to be a dependency of the Code Coverage Analyzer.

9.3.4 Installation

This tool is a a build-in platform tool. There is nothing else to install in the platform to be able to use it.

9.3.5 Use

9.3.5.1 MicroEJ Tool

A MicroEJ tool is available to launch the Code Coverage Analyzer tool. The tool name is **Code Coverage Analyzer** .

9.3.5.2 Category: Code Coverage

The screenshot shows the 'Code Coverage' configuration window. It features a tab labeled 'Code Coverage'. The main configuration area includes a text box for '*cc files folder:' with a 'Browse...' button to its right. Below this is a 'Classes filter' section. This section contains two sub-sections: 'Includes:' and 'Excludes:'. Each sub-section has a large text box for input and three buttons to its right: 'Add...', 'Edit...', and 'Remove'.

9.3.5.2.1 Option(browse): *.cc files folder

Default value: (empty)

Description:

Specify a folder which contains the cc files to process (*.cc).

9.3.5.2.2 Group: Classes filter

9.3.5.2.2.1 Option(list): Includes

Default value: (empty)

Description:

List packages and classes to include to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

packageA.packageB.*: includes all classes which are in package packageA.packageB

packageA.packageB.className: includes the class packageA.packageB.className

9.3.5.2.2.2 Option(list): Excludes

Default value: (empty)

Description:

List packages and classes to exclude to code coverage report. If no package/class is specified, all classes found in the project classpath will be analyzed.

Examples:

packageA.packageB.*: excludes all classes which are in package packageA.packageB

packageA.packageB.className: excludes the class packageA.packageB.className

9.3.5.3 Operation

Two levels of code analysis are provided, the Java level and the bytecode level, as well as a view of the fully or partially covered classes and methods. From the HTML report index, just use hyper-links to navigate into the report and source / bytecode level code.

9.4 Heap Dumper**9.4.1 Introduction**

The heap is a memory area used to hold Java objects created at runtime. Objects persist in the heap until they are garbage collected. An object becomes eligible for garbage collection when there are no longer any references to it from other objects.

Heap Dumper is a tool that takes a snapshot of the heap (.heap dump file) and analyzes its contents to help

- finding memory leaks,
- browsing objects instances,
- optimizing the heap usage (using immortals or immutables).

It works only on SimJPF MicroEJ.

9.4.2 Functional Description

Refer to Section 12 , SimJPF MicroEJ launcher options to know how to activate SimJPF Heap Dumper option.

9.4.3 Dependencies

No dependency is required.

9.4.4 Installation

This tool is a a built-in platform tool. There is nothing else to install in the platform to be able to use it.

9.4.5 Use

The heap dump files are generated during the calls of the System.gc() method. To use the heap dumper, you need to call this method in your code. Each call of System.gc() method performs a dump of the current state of the heap and generates a heap dump file that represent a snapshot of the heap at this moment.

The heap dump file generated contains the list of all instances of both class and array types that exist in the heap. For each instance it records:

- the time at which the instance was created,
- the thread that created it,
- the method that created it.

For instances of class types, it also records:

- the class,
- the values in the instance's non-static fields.

For instances of array types, it also records:

- the type of the contents of the array,
- the contents of the array.

For each referenced class type it records the values in the static fields of the class.

9.5 Test Suite

9.5.1 Definition

The MicroEJ Test-Suite is an engine made for validating any development project using automatic testing. The MicroEJ Test-Suite engine allow the user to test any kind of projects within the configuration of a generic ant file.

9.5.2 Dependencies

No dependency is required.

9.5.3 Installation

This tool is a built-in platform tool. There is nothing else to install in the platform to be able to use it.

9.5.4 Using the MicroEJ Test-Suite Ant tasks

Multiple Ant tasks are available in the testsuite-engine provided jar:

- `testsuite` allows the user to run a given testsuite and to retrieve an XML report document in a JUnit format.
- `javaTestsuite` is a subtask of the `testsuite` task, used to run a specialized testsuite for Java (will only run Java classes).
- `htmlReport` is a task which will generate an HTML report from a list of JUnit report files.

9.5.4.1 The testsuite task

This task have some mandatory attributes to fill:

- `outputDir`: the output folder of the test-suite. The final report will be generated at `[outputDir]/[label]/[reportName].xml`, see the `testsuiteReportFileProperty` and `testsuiteReportDirProperty` attributes.
- `harnessScript`: the harness script must be an Ant script and it is the script which will be called for each test by the test-suite engine. It is called with a `basedir` located at output location of the current test. The test-suite engine will provide to it some properties giving all the informations to start the test:
 - `testsuite.test.name`: The output name of the current test in the report. Default value is the relative path of the test. It can be manually set by the user. More details on the output name are available in the section *Specific custom properties*.
 - `testsuite.test.path`: The current test absolute path in the filesystem.
 - `testsuite.test.properties`: The absolute path to the custom properties of the current test (see the property `customPropertiesExtension`)
 - `testsuite.common.properties`: The absolute path to the common properties of all the tests (see the property `commonProperties`)

- `testsuite.report.dir`: The absolute path to the directory of the final report.

Some attributes are optional, and if not set by the user, a default value will be attributed.

- `timeOut`: the time in seconds before any test is considered as unknown. Set it to 0 to disable the time-out. Will be defaulted as 60.
- `verboseLevel`: the required level to output messages from the test-suite. Can be one of those values: error, warning, info, verbose, debug. Will be defaulted as info.
- `reportName`: the final report name (without extension). Default value is `testsuite-report`.
- `customPropertiesExtension`: the extension of the custom properties for each test. For instance, if it is set to `.options`, a test named `xxx/Test1.class` will be associated with `xxx/Test1.options`. If a file exists for a test, the property `testsuite.test.properties` is set with its absolute path and given to the harnessScript. By default, custom properties extension is `.properties`.
- `commonProperties`: the properties to apply to every test of the test-suite. Those options might be overridden by the custom properties of each test. If this option is set and the file exists, the property `testsuite.common.properties` is set to the absolute path to this file to the harnessScript. By default, there is not any common properties.
- `label`: the build label. Will be generated as a timestamp by the test-suite if not set.
- `productName`: the name of the current tested product. Default value is `TestSuite`.
- `jvm`: the location of your Java VM to start the testsuite (the harnessScript is called as is: `[jvm] [...] -buildfile [harnessScript]`). Will be defaulted as your `java.home` location if the property is set, or to `java`.
- `jvmargs`: the arguments to pass to the Java VM started for each test.
- `testsuiteReportFileProperty`: the name of the Ant property in which is stored the path of the final report. Default value is `testsuite.report.file` and path is `[outputDir]/[label]/[reportName].xml`
- `testsuiteReportDirProperty`: the name of the Ant property in which is store the path of the directory of the final report. Default value is `testsuite.report.dir` and path is `[outputDir]/[label]`
- `testsuiteResultProperty`: the name of the Ant property in which you want to have the result of the test-suite (true or false), depending if every tests successfully passed the test-suite or not. Ignored tests do not affect this result.

Finally, you have to give as nested element the path containing the tests.

- `testPath`: containing all the file of the tests which will be launched by the test-suite.
- `testIgnoredPath` (optional): Any test in the intersection between `testIgnoredPath` and `testPath` will be executed by the test-suite, but will not appear in the JUnit final report. It will still generate a JUnit report for each test, which will allow the HTML report to let them appears as "ignored" if it is generated. Mostly used for known bugs which are not considered as failure but still relevant enough to appears on the HTML report.

9.5.4.2 The javaTestsuite task

This task extends the testsuite task, specializing the test-suite to only start real Java class. This task will retrieve the classname of the tests from the classfile and will provide new properties to the harness script:

- `testsuite.test.class`: The classname of the current test. The value of the property `testsuite.test.name` is also set to the classname of the current test.

- `testsuite.test.classpath`: The classpath of the current test.

9.5.4.3 The *htmlReport* task

This task allow the user to transform a given path containing a sample of JUnit reports to an HTML detailed report. Here is the attributes to fill:

- A nested fileset containing all the JUnit reports of each test. Take care to exclude the final JUnit report generated by the testsuite.
- A nested element report
 - `format`: The format of the generated HTML report. Must be `noframes` or `frames`. When `noframes` format is choosen, a standalone HTML file is generated.
 - `todir`: The output folder of your HTML report.
 - The report tag accepts the nested tag param with name and expression attributes. These tags can pass XSL parameters to the stylesheet. The built-in stylesheets support the following parameters:
 - `PRODUCT`: the product name that is displayed in the title of the HTML report.
 - `TITLE`: the comment that is displayed in the title of the HTML report.

Tip: It is advised to set the format to `noframes` if your testsuite is not a Java testsuite. If the format is set to `frames`, with a non-Java MicroEJ Test-Suite, the name of the links will not be relevant because of the non-existancy of packages.

9.5.5 JUnit report generation

The test-suite engine will generate a final report in a junit format, which can be easily read in Eclipse via the JUnit view. This format allow the user to have a quick and easy review of the test-suite results.

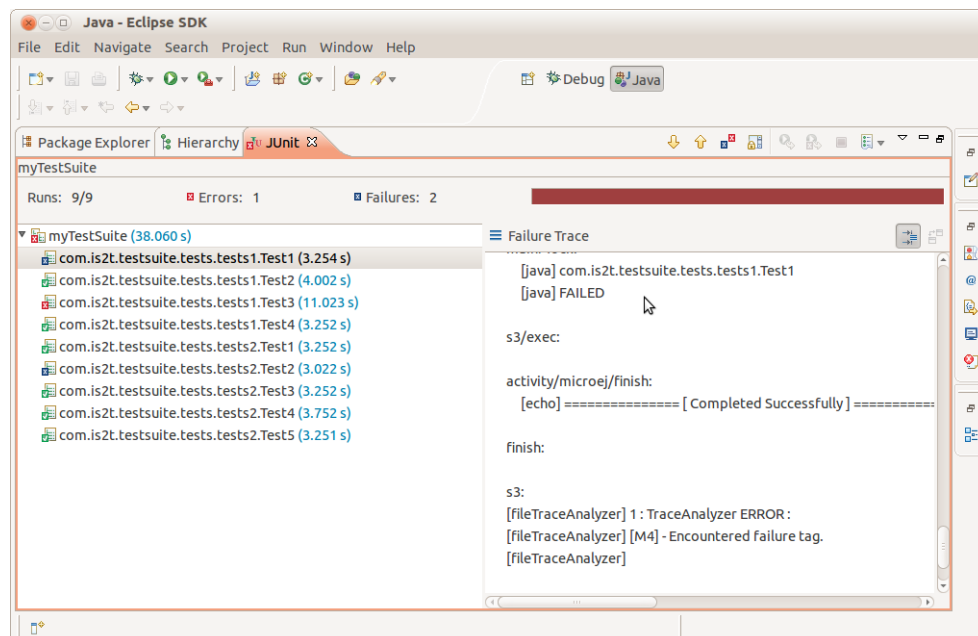


Figure 9.6. JUnit final report example

Every test might end in three different states.

- **Error**: When the `time out` is reached, and the test has not finished to run, the current test result will be considered as unknown, which will result as error in the JUnit report.

- Failure: When the harness script fails, the current test result will be considered as failure.
- Passed: When the harness script does not fail and ends within the given time out, the current test result will be considered as passed.

The execution trace of any test will be saved into the report, but the Junit view will only show the trace of a test which ended as an error or as a failure.

JUnit format is an XML format and the final report respects the following DTD:

```
<!ELEMENT testsuite (testcase+)>
<!--ATTLIST testsuite
  errors CDATA #REQUIRED
  failures CDATA #REQUIRED
  hostname CDATA #REQUIRED
  ignored CDATA #REQUIRED
  name CDATA #REQUIRED
  started CDATA #REQUIRED
  tests CDATA #REQUIRED
  time CDATA #REQUIRED
  timestamp CDATA #REQUIRED
-->

<!--ELEMENT testcase ((failure|error|system-out|system-err)?)>
<!--ATTLIST testcase
  classname CDATA #REQUIRED
  name CDATA #REQUIRED
  time CDATA #REQUIRED
-->

<!--ELEMENT failure (#PCDATA)>
<!--ELEMENT error (#PCDATA)>
<!--ELEMENT system-out (#PCDATA)>
<!--ELEMENT system-err (#PCDATA)>
```

9.5.6 Example - Deploying an Ant test-suite

The goal of this section is to describe the steps to deploy a test suite environment on a simple application. For this example, we will start a simple Ant Test Suite.

9.5.6.1 Tests cases

For this example, we have two Ant tests, Test1.xml and Test2.xml.

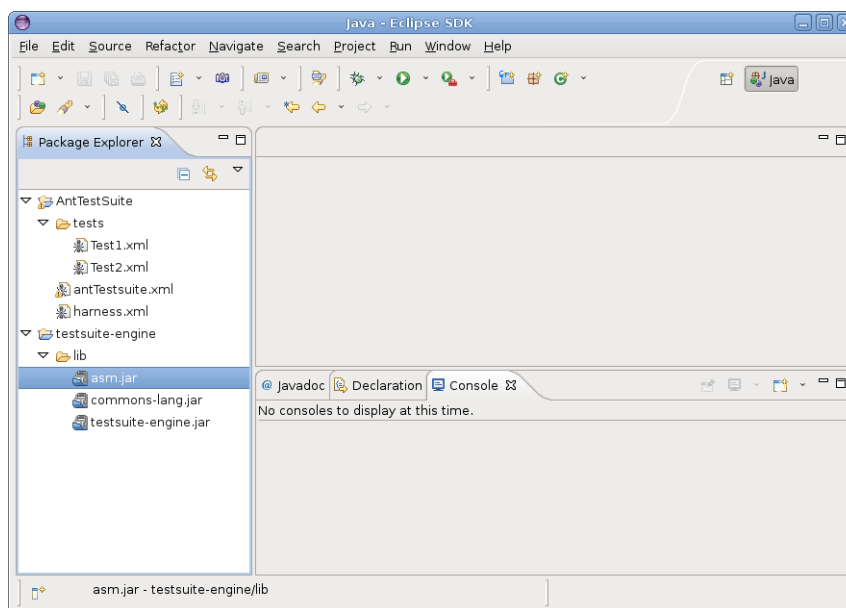


Figure 9.7. Example Ant - The tree files

1 Test1.xml

```
<project name="test1" default="build" >
  <target name="build" >
    <echo message="Test OK"/>
  </target>
</project>
```

2 Test2.xml

```
<project name="test2" default="build" >
  <target name="build" >
    <fail message="Test NOK"/>
  </target>
</project>
```

9.5.6.2 Creating the test harness

The harness.xml script will just run the given test. Here is the content of this script:

```
<project name="harness" default="runTest">
  <target name="runTest">
    <!-- Just execute the Ant script -->
    <ant antfile="${testsuite.test.path}"/>
  </target>
</project>
```

9.5.6.3 Creating the MicroEJ Test-Suite launch

The antTestsuite.xml script will define the MicroEJ Test-Suite tasks, execute the testsuite and generate an HTML report.

9.5.6.3.1 The testsuite tasks definition

The definition of the testsuite tasks is made by loading the antlib defined by the MicroEJ Test-Suite. The testsuite XML namespace is defined in the root tag of the Ant project.

```
<project name="antTestsuite" default="run"
  xmlns:testsuite="antlib:com.is2t.testsuite.ant">
  ...

  <target name="testsuiteDefinition">
    <property name="testsuite.lib.dir" location="../../testsuite-engine/lib"
      description="Path to the testsuite engine jars."/>

    <!-- Define testsuite tasks -->
    <taskdef uri="antlib:com.is2t.testsuite.ant" resource="com/is2t/testsuite/ant/
antlib.xml">
      <classpath>
        <fileset dir="${testsuite.lib.dir}" includes="*.jar"/>
      </classpath>
    </taskdef>
  </target>
</project>
```

9.5.6.3.2 The testsuite:testsuite task

As explained on the section Using the test-suite Ant tasks, we will use the testsuite:testsuite task. Here is the call of the task:

```
<testsuite:testsuite outputDir="results~" harnessScript="harness.xml">
  <testPath>
    <fileset dir="tests" includes="*.xml"/>
  </testPath>
</testsuite:testsuite>
```

Here is the JUnit report generated by the launch:

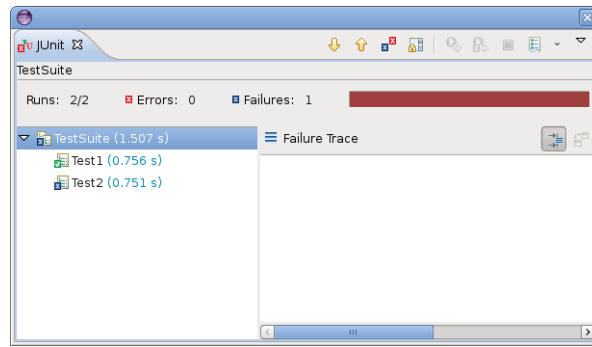


Figure 9.8. Example Ant - Final JUnit report

9.5.6.3.3 The testsuite:htmlReport task

The testsuite:htmlReport task is used to generate the HTML report. A fileset containing the JUnit reports of each tests is given to the task.

```
<testsuite:htmlReport>
<fileset dir="${testsuite.report.dir}"> <!-- The 'testsuite.report.dir' property has
  been defined by the 'testsuite:testsuite' task -->
  <include name="**/*.xml"/> <!-- include unary reports -->
  <exclude name="*.xml"/> <!-- exclude global report -->
</fileset>
<report format="noframes" todir="${testsuite.report.dir}"/>
</testsuite:htmlReport>
```

Here is the HTML report (noframes) generated by the launch:

Testsuite Results:

Summary

Tests	Failures	Errors	Ignored	Success rate	Time
2	1	0	0	50.00%	1.507

Assertions	Failures	Success	Success Rate
0	0	0	NaN

Note: **failures** are anticipated and checked for with assertions while **errors** are unanticipated.

Note: **ignored** tests are executed but not counted on the success rate.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Ignored	Time(s)	Time Stamp	Host
	2	0	1	0	1.507	1388153526037	local

Package

Name	Tests	Errors	Failures	Ignored	Time(s)	Time Stamp	Host
Test1	1	0	0	0	0.756	1388153526037	local
Test2	1	0	1	0	0.751	1388153526813	local

Figure 9.9. Example Ant - Final HTML report

9.5.7 Example - Deploying a Java test-suite

The goal of this section is to describe the steps to deploy a test suite environment on a simple application. For this example, we will start a simple MicroEJ application Test Suite on a IS2T Platform.

9.5.7.1 The MicroEJ environment

This example requires to have a MicroEJ environment installed, and at least a platform on your MicroEJ repository.

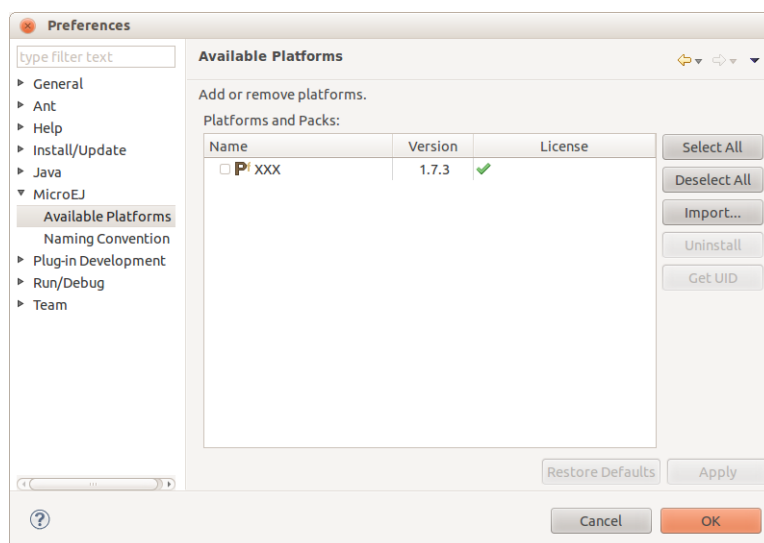


Figure 9.10. Example Java - Available platforms

9.5.7.2 Preview of the test-suite

For this example, we have four Java tests.

- Test1.java, which will fail.
- Test2.java, which will pass.
- Test3.java, which will do nothing.
- Test4.java, which will do infinitely.
- Test5.java, which will fail but is ignored.

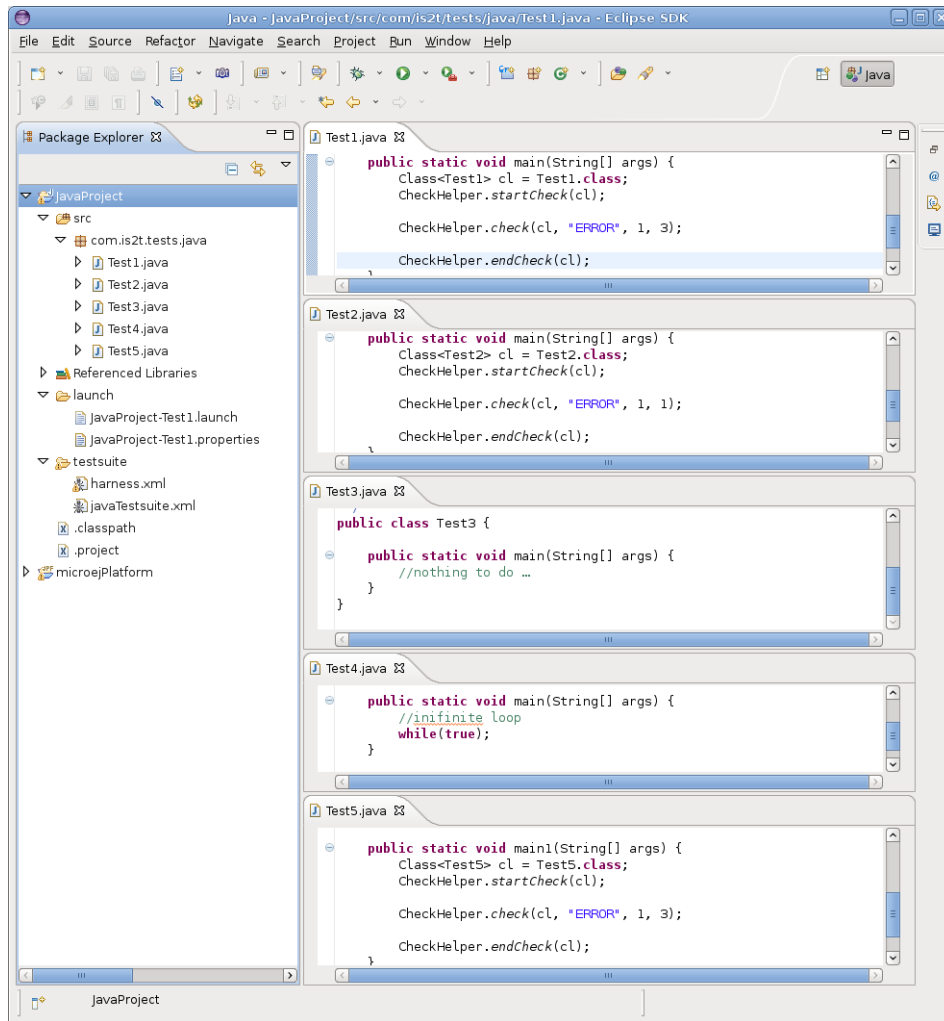


Figure 9.11. Example Java - The tree files

9.5.7.3 Creating the MicroEJ application launch

First, create a MicroEJ application launch, to retrieve all the common properties which are platform-dependants and common to every tests that will run through the test-suite. To do so, open the Eclipse menu **Run** → **Run Configuration...** and double click on MicroEJ Application.

Configure your launch as the following screenshots, for one of the tests, for example Test1.java. Do not forget to check in the Common Tab to save the launch as a Shared file to be able to retrieve a property file from this launch.

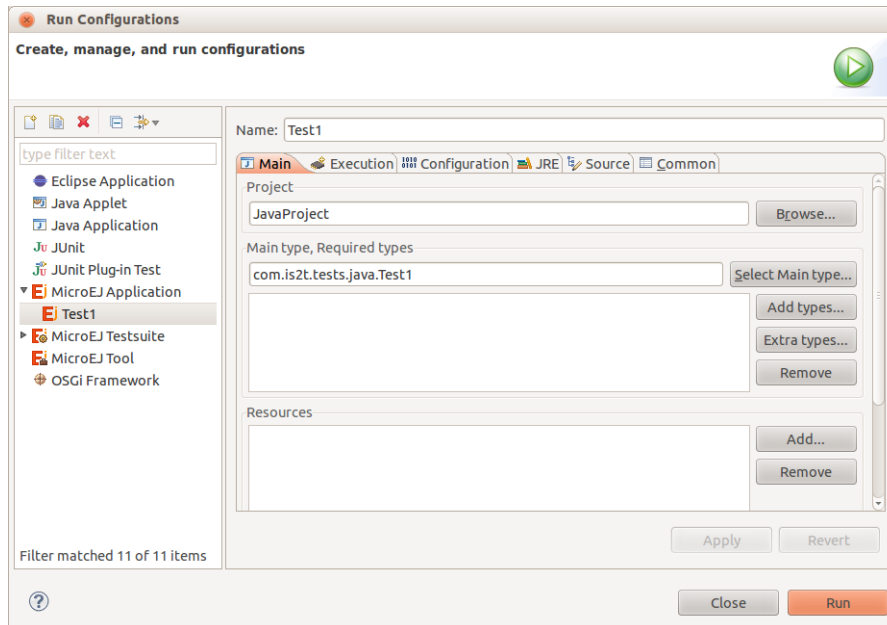


Figure 9.12. Example Java - Main Tab

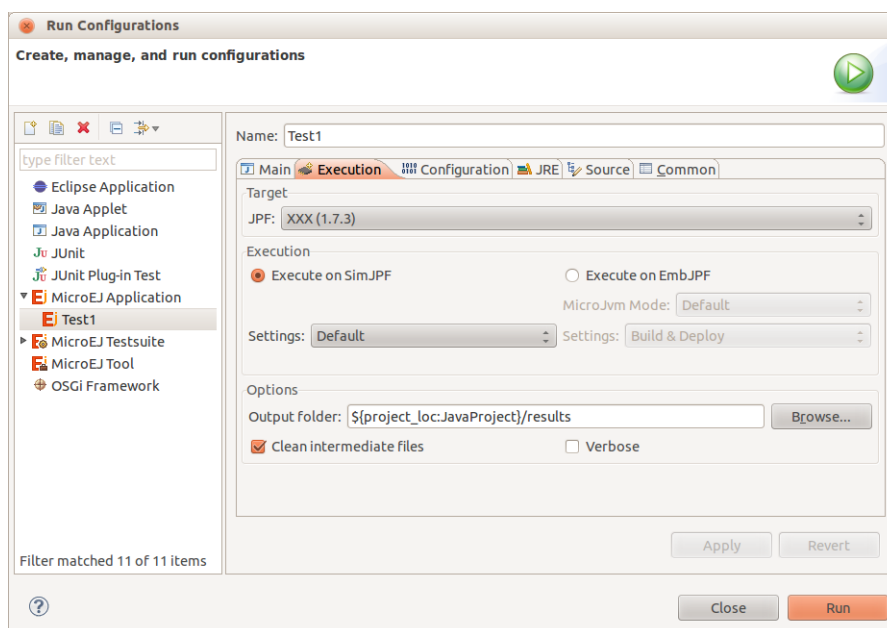


Figure 9.13. Example Java - Execution Tab

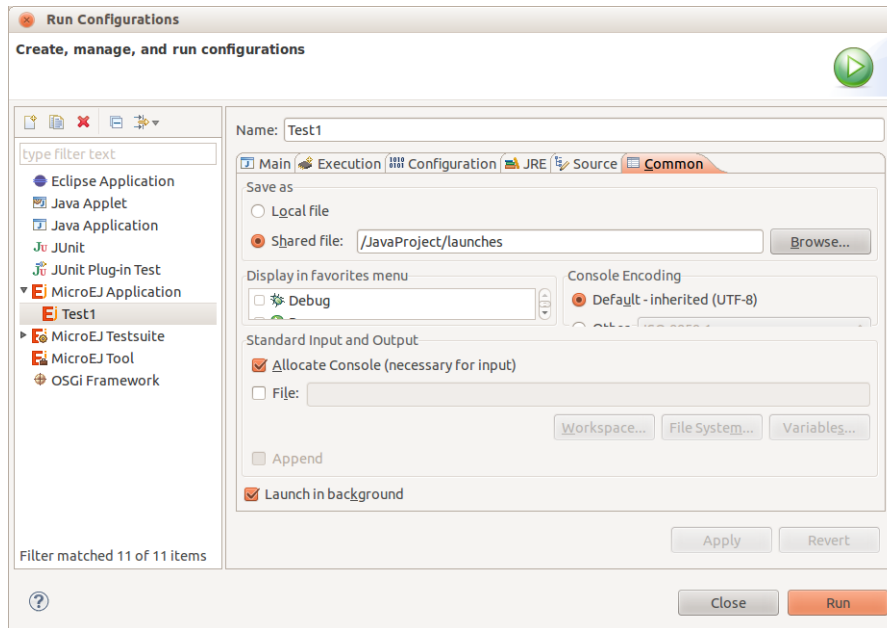


Figure 9.14. Example Java - Common Tab

Looking that we set up the test to be ran on simJPF, the script called by the launch will be `s3Default.microejLaunch`. When starting this launch, the expected output in the console are the following line. Looking we started the launch on `Test1.java`, we can see that the word "FAILED" as been wrote in the output.

```
===== [  INITIALIZATION STAGE  ] =====
===== [  LAUNCHING S3          ] =====
FAILED
LAUNCH SUCCESSFUL
```

9.5.7.4 The MicroEJ Test-Suite harness

The harness script is the script that will be started for every test. For this example, we will use the `File Trace Analyzer` provided by the MicroEJ Platform. The `File Trace Analyzer` is a tool allowing the user to parse a file searching for defined tags. Per default, it will search the following tags: `PASSED` and `FAILED`.

The harness consists in the following steps:

- Try to load the customs properties of the current test, located in the file saved in the property `testsuite.test.properties` sent by the test-suite engine. It try to loads them before loading the common properties, to override them if any conflicts.
- Try to load the common properties of all the tests, located in the file saved in the property `testsuite.common.properties` sent by the test-suite engine.
- Create a temporary file to store the standard output.
- Start the MicroEJ launch script `s3Default.microejLaunch` provided by the MicroEJ platform.
- Analyze the trace outputed in the temporary file to fail if the success tag is not encountered.

```

<property file="${testsuite.test.properties}"/>
<property file="${testsuite.common.properties}"/>

<fail unless="platform.dir" message="Please set the 'platform.dir' property."/>

<!-- Import the script that defines the trace analyzer tasks -->
<import file="${platform.dir}/scripts/traceAnalyzerDefinition.xml"/>

<target name="runTest" depends="traceAnalyzer/definition">

  <!-- Create and clean microej.io.tmpdir (done by the workbench but not done when
  running from testsuite engine) -->
  <delete failonerror="false" dir="${microej.io.tmpdir}"/>
  <mkdir dir="${microej.io.tmpdir}"/>

  <!-- Prepare the file to save the trace -->
  <tempfile property="trace.file" prefix="trace" suffix=".txt"
  destdir="${microej.io.tmpdir}" deleteonexit="true"/>
  <record name="${trace.file}" action="start" />

  <!-- Run underlying the launch -->
  <ant antfile="${platform.dir}/scripts/s3Default.microejLaunch">
    <property name="application.classpath"
    value="${testsuite.test.classpath}${path.separator}${application.classpath}"/>
    <property name="application.main.class" value="${testsuite.test.class}"/>
    <property name="output.dir" value="${testsuite.report.dir}/bin"/>
    <property name="basedir" value="${platform.dir}/scripts"/>
  </ant>

  <!-- Trace has ended -->
  <record name="${trace.file}" action="stop" />

  <!-- Analyze trace. Trace is fully generated: stop when EOF is reached -->
  <traceAnalyzer:fileTraceAnalyzer stopEOFReached="true" traceFile="${trace.file}"/>

  <!-- If the script reaches this point, test result is success -->
</target>

```

9.5.7.5 Creating the MicroEJ Test-Suite launch

The `javaTestsuite.xml` script will define the MicroEJ Test-Suite tasks, execute the testsuite and generate an HTML report.

9.5.7.5.1 Definition of the tests to launch

A path called `testsuite.tests.paths` is defined with the tests to execute and another path called `testsuite.ignored.tests.paths` is defined with the tests to ignore.

```

<path id="testsuite.tests.paths">
  <fileset dir="${ant.dir.runTestsuite}/../bin/">
    <include name="**/T*.class"/>
  </fileset>
</path>

<path id="testsuite.ignored.tests.paths">
  <fileset dir="${ant.dir.runTestsuite}/../bin/">
    <include name="**/T*5.class"/>
  </fileset>
</path>

```

9.5.7.5.2 The testsuite:javaTestsuite task

The `testsuite:javaTestsuite` task is defined by calling the `testsuite/definition` target defined in a script provided by the MicroEJ Platform. This task takes the MicroEJ Launch properties file as argument.

```

<!-- Import the script that defines the testsuite tasks -->
<import file="${platform.dir}/scripts/testsuiteDefinition.xml"/>

<target name="run" depends="testsuite/definition">

  <!-- Launch test suite -->
  <testsuite:javaTestsuite
    outputDir="${ant.dir.runTestsuite}/../results~"
    harnessScript="${ant.dir.runTestsuite}/harness.xml"
    commonProperties="${microej.launch.propertyfile}"
  >
    <testPath refid="testsuite.tests.paths"/>
    <testIgnoredPath refid="testsuite.ignored.tests.paths"/>
  </testsuite:javaTestsuite>

  ...
</target>

```

Here is the JUnit report generated by the launch:

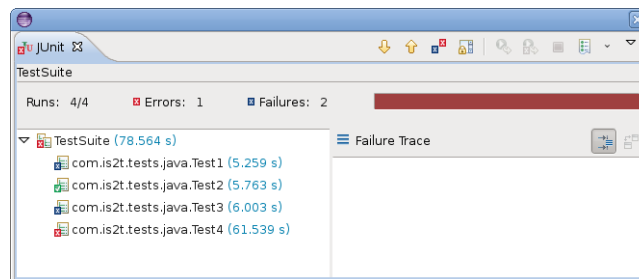


Figure 9.15. Example Java - Final JUnit report

9.5.7.5.3 The testsuite:htmlReport task

The testsuite:htmlReport task is used to generate the HTML report. A fileset containing the JUnit reports of each tests is given to the task.

```

<!-- Generate HTML report -->
<testsuite:htmlReport>
  <fileset dir="${testsuite.report.dir}">
    <include name="**/*.xml"/> <!-- include unary reports -->
    <exclude name="*.xml"/> <!-- exclude global report -->
  </fileset>
  <report format="noframes" todir="${testsuite.report.dir}"/>
</testsuite:htmlReport>

```

Here is the HTML report (noframes) generated by the launch:

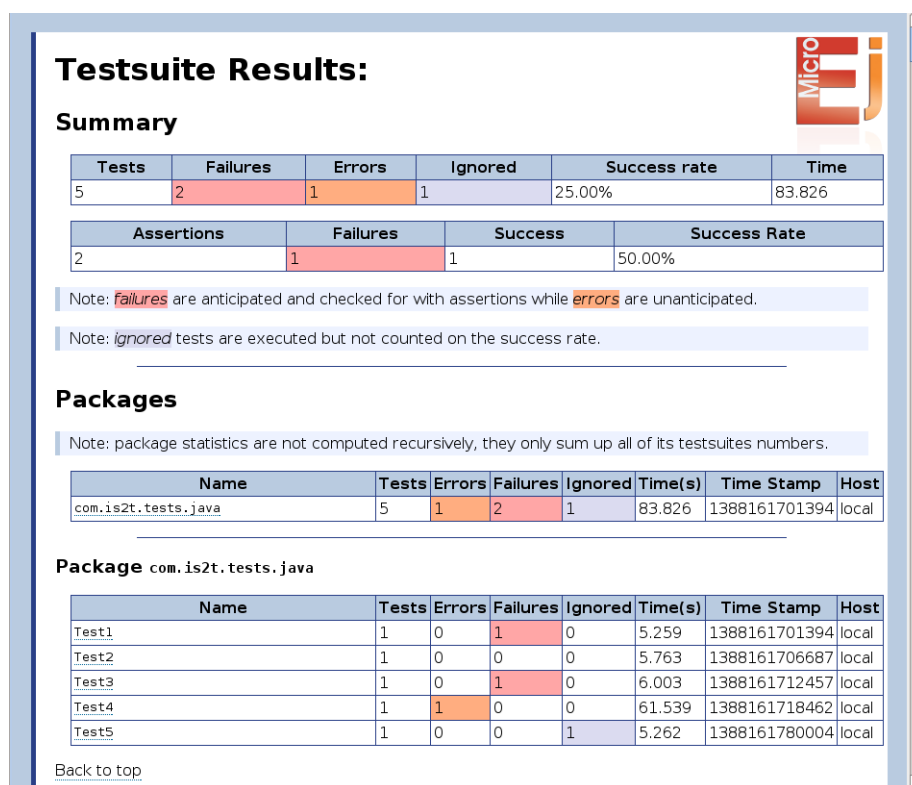


Figure 9.16. Example Java - Final HTML report

9.5.8 Using the trace analyzer

This section will shortly explain how to use the Trace Analyzer. The MicroEJ Test-Suite comes with an archive containing the Trace Analyzer which can be used to analyze the output trace of an application. It can be used from different forms;

- The FileTraceAnalyzer will analyze a file and research for the given tags, failing if the success tag is not found.
- The SerialTraceAnalyzer will analyze the data from a serial connection.

9.5.8.1 The TraceAnalyzer tasks options

Here is the common options to all TraceAnalyzer tasks:

- successTag: the regular expression with is synonym of success when found (by default . *PASSED. *).
- failureTag: the regular expression with is synonym of failure when found (by default . *FAILED. *).
- verboseLevel: int value between 0 and 9 to define the verbose level.
- waitingTimeAfterSuccess: waiting time (in s) after success before closing the stream (by default 5).
- noActivityTimeout: timeout (in s) with no activity on the stream before closing the stream. Set it to 0 to disable timeout (default value is 0).
- stopEOFReached: boolean value. Set to true to stop analyzing when input stream EOF is reached. If false, continue until timeout is reached (by default false).
- onlyPrintableCharacters: boolean value. Set to true to only dump ASCII printable characters (by default false).

9.5.8.2 The FileTraceAnalyzer task options

Here is the specific options of the FileTraceAnalyzer task:

- `traceFile`: path to the file to analyze.

9.5.8.3 The *SerialTraceAnalyzer* task options

Here is the specific options of the *SerialTraceAnalyzer* task:

- `port`: the comm port to open.
- `baudrate`: serial baudrate (by default 9600).
- `databits`: databits (5|6|7|8) (by default 8).
- `stopBits`: stopbits (0|1|3 for (1_5)) (by default 1).
- `parity`: none | odd | event (by default none).

9.5.9 Appendix

The goal of this section is to explain some tips and tricks that might be useful in your usage of the test-suite engine.

9.5.9.1 Specific custom properties

Some custom properties are specifics and retrieved from the test-suite engine in the custom properties file of a test.

- The `testsuite.test.name` property is the output name of the current test. Here are the steps to compute the output name of a test:
 - If the custom properties are enabled and a property named `testsuite.test.name` is found on the corresponding file, then the output name of the current test will be set to it.
 - Otherwise, if the running MicroEJ Test-Suite is a Java testsuite, the output name is set to the class name of the test.
 - Otherwise, from the path containing all the tests, a common prefix will be retrieved. The output name will be set to the relative path of the current test from this common prefix. If the common prefix cuts the name of the test, then the output name will be set to the name of the test.
 - Finally, if multiples tests have the same output name, then the current name will be followed by `_xxx`, an underscore and an integer.
- The `testsuite.test.timeout` property allows the user to redefine the time out for each test. If it is negative or not an integer, then global timeout defined for the MicroEJ Test-Suite is used.

10 Build Support

MicroEJ provides additional modules which allows to configure the platform itself and the external platform projects useful to build the platform.

10.1 Board Support Package

10.1.1 Introduction

When the Java platform is built, the user is able to compile a Java application onto. The result of this compilation is not sufficient. A third-party C project is required to obtain the final binary file to flash on a board.

This third-party C project is usually configured to target only one board. It contains some C files, header directories, C libraries etc. Thanks to this C project the user is able to build (compile and link) a binary file which contains the specific MCU and board libraries, the Java libraries and the Java application.

This module configures the third-party project updating the third-party C-IDE project file, adding some C libraries and filling some header directories.

10.1.2 XML File

The Java platform has to have some information to know how the board project (the board support package) is. This information is useful to be able to build a Java application compatible with the board support package.

A XML file is so required to configure the Java platform. The name of this file must be `bsp.xml`. It has to be stored in the module configuration folder (see “Installation”).

This file must start with the node `<bsp>`. It can contain several time this kind of line: `<nativeName="A_LLAPI_NAME" nativeImplementation name="AN_IMPLEMENTATION_NAME"/>` where:

- `A_LLAPI_NAME` refers to a Low Level API native name. It is specific to the MicroEJ C library which provides the Low Level API.
- `AN_IMPLEMENTATION_NAME` refers to the implementation name of the Low Level API. It is specific to the Board Support Package, more specifically to the C file which does the link between the MicroEJ C library and the C driver.

Example:

```
<bsp>
  <nativeImplementation name="LLCOMM_DEFAULT" nativeName="LLCOMM_BUFFERED_CONNECTION"/>
</bsp>
  <nativeImplementation name="LLDISPLAY_STM32x0GEVAL" nativeName="LLDISPLAY_COPY"/>
  <nativeImplementation name="TOUCH" nativeName="LLINPUT_DEVICE"/>
</bsp>
```

10.1.3 Dependencies

No dependency.

10.1.4 Installation

This is an additional and optional module useful to configure the board C project. Install it to configure the project automatically during the platform creation.

In the platform configuration file, check `Miscellaneous > Board Support Package` to install it. When checked, the properties file `bsp > bsp.properties` is required during platform creation to configure the module.

The properties file have to / can contain the following properties:

- `project.file` [optional, default value is "" (*empty*)]: Defines the full path of the C project file. This file will be updated with the platform libraries. If not set or empty, no C project is updated.
- `project.libs.group.name` [optional, default value is "" (*empty*)]: Defines the libraries group name of the C project file. This property is required if property `project.file` is set.
- `project.includes.output.dir` [optional, default value is "" (*empty*)]: Defines the full path of the C project other header files (*.h) output directory. All platform header files (*.h) will be copied into. If not set or empty no header platform files is copied.

The XML configuration file (see “XML File”) `bsp > bsp.xml` is also required during platform creation to configure the Java platform.

10.1.5 Use

This module is automatically used during the platform creation.

10.2 Java Examples

10.2.1 Introduction

The Java platform can contain some Java examples. These Java examples will be used by the Java platform user to launch a Java example on it. This feature is very useful when the Java platform contain a custom Java library. A Java example allows to the Java platform user to understand quickly how the Java library runs.

This module allows to install some Java examples into the Java platform.

10.2.2 Dependencies

No dependency.

10.2.3 Installation

In the platform configuration file, check `Miscellaneous > Java Examples` to install some Java examples into the Java platform.

10.2.4 Use

This module is automatically used during the platform creation.

11 Simulation

11.1 Introduction

The JPF provides an accurate JPF simulator that runs on workstations, called the SimJPF. Applications execute in an almost identical manner on both the workstation and on target devices. The SimJPF features IO simulation, JDWP debug coupled with Eclipse, accurate Java heap dump, and an accurate Java scheduling policy (the same as the embedded one)⁴.

11.2 Functional Description

In order to simulate external stimuli that come from the native world (that is, "the C world"), the SimJPF has a Hardware In the Loop interface, HIL, which simulates on the workstation "Java-to-C calls". All Java-to-C calls are rerouted to an HIL engine. Indeed HIL is a replacement for the [SNIGT] interface.

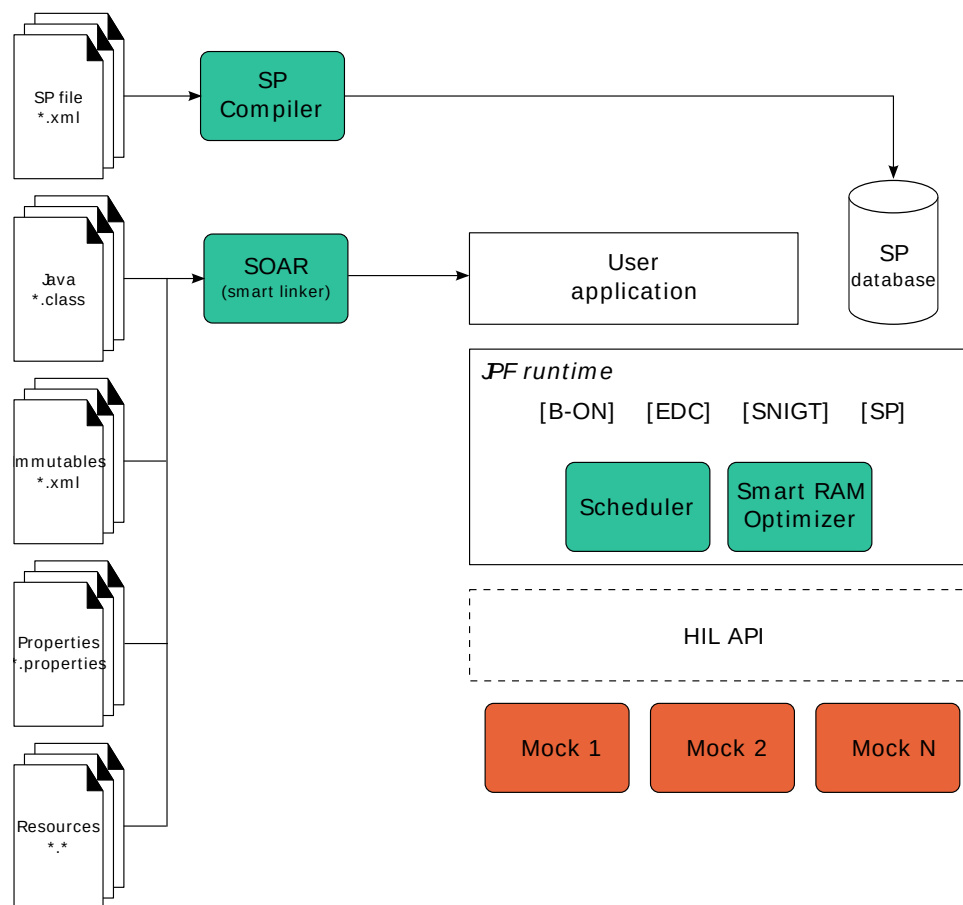


Figure 11.1. The HIL connects the SimJPF to the workstation.

The "simulated C world" is made of Mocks that simulate native code such as drivers and any other kind of C libraries, so that the Java application can behave the same as the device using the EmbJPF.

The SimJPF and the HIL are two processes that runs in parallel: the communication between them is through a socket connection. Mocks run inside the process that runs the HIL engine.

⁴ Only the execution speed is not accurate. The simulator speed can be set to match the average embedded JPF speed in order to adapt the simulator speed to the desktop speed.

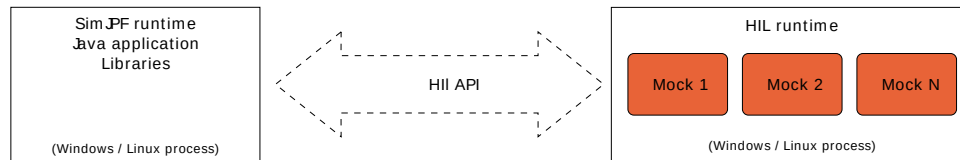


Figure 11.2. A SimJPF socket-connected to its HIL engine.

11.3 Mock

11.3.1 Introduction

The HIL engine for the JPF is a Java-based engine that runs Java mocks. A mock is a jar file containing some Java classes that simulate natives for the simulator. Mocks allow applications to be run unchanged in the simulator while still (apparently) interacting with native code.

11.3.2 Functional Description

As with [SNIGT], HIL is responsible for finding the method to execute as a replacement for the native Java method that the SimJPF tries to run. Following the [SNIGT] philosophy, the matching algorithm uses a naming convention. In fact, when using the provided HIL engine, the same name is used on both sides. When a native method is called in the SimJPF, it requests the HIL engine to execute it. The corresponding mock executes the method and provides the result back to the SimJPF.

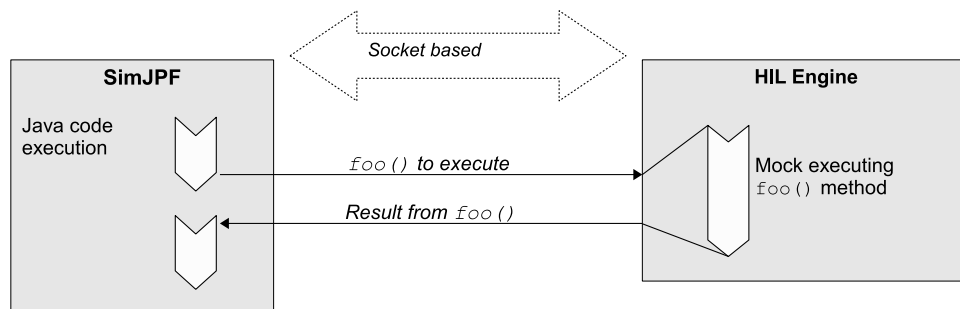


Figure 11.3. The SimJPF executes a native Java method `foo()`.

11.3.3 SimJPF Interface: Mocks Design Support

11.3.3.1 Interface

The SimJPF interface is defined by static methods on the Java class `com.is2t.hil.NativeInterface`.

11.3.3.2 Array Type Arguments

Both [SNIGT] and HIL allow arguments that are arrays of base types. By default the content of an array is NOT sent over to the mock. An "empty copy" is sent by the HIL engine, and the content of the array must be explicitly fetched by the mock. The array within the mock can be modified using regular assignment. Then to apply these changes in the SimJPF, the modifications need to be flushed back. There are two methods provided to support fetch and flush between the SimJPF and the HIL:

- `refreshContent`: initializes the array argument from the content of its SimJPF counterpart.
- `flushContent`: propagates (to the SimJPF) the content of the array that is used within the HIL engine.

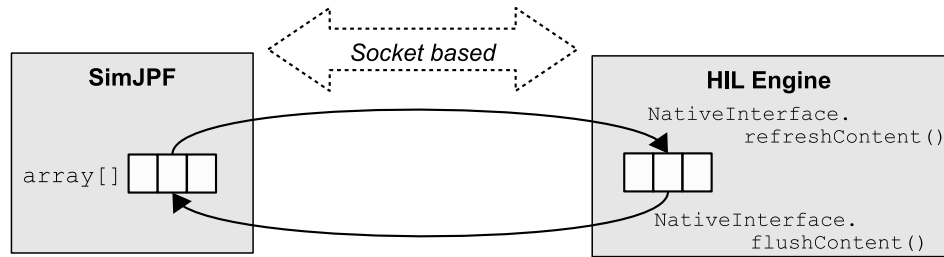


Figure 11.4. An array and its counterpart in the HIL engine.

Below is a typical usage.

```
public static void foo(char[] chars, int offset, int length){
    NativeInterface ni = HIL.getInstance();
    //inside the mock
    ni.refreshContent(chars, offset, length);
    chars[offset] = 'A';
    ni.flushContent(chars, offset, 1);
}
```

Figure 11.5. Typical usage of HIL engine.

11.3.3.3 Blocking Native Methods

Some native methods block until an event has arrived [SNIGT]. Such behavior is implemented in a mock using the following three methods:

- `suspendCurrentJavaThread(long timeout)`: tells the SimJPF that the green thread should block after returning from the current native. This method does not block the mock execution. The green thread is suspended until either a mock thread calls `resumeJavaThread` or if the specified amount of milliseconds has elapsed.
- `resumeJavaThread(int id)`: resumes the green thread with given ID. If the thread is not suspended, the resume stays pending and the next call to `suspendCurrentJavaThread` will not block the thread.
- `getCurrentJavaThreadID()`: retrieves the ID of the current Java thread. This ID must be given to the `resumeJavaThread` method in order to resume the green thread execution.

```
public static byte[] Data = new byte[BUFFER_SIZE];
public static int DataLength = 0;

//Mock native method
public static void waitForData(){
    NativeInterface ni = HIL.getInstance();
    //inside the mock
    //wait until the data is received
    setWaitingThread(ni.getCurrentJavaThreadID());
    if(DataLength == 0){
        ni.suspendCurrentJavaThread(0);
    }
}

//Mock data reader thread
public static void notifyDataReception()
{
    NativeInterface ni = HIL.getInstance();
    DataLength = readFromInputStream(Data);
    ni.resumeJavaThread(getWaitingThread());
}
```

Figure 11.6. Suspend/resume Java Threads example

11.3.3.4 Resource Management

In Java, every class can play the role of a small read-only file system root: the stored files are called "Java resources" and are accessible using a path as a String.

The SimJPF interface allows the retrieval of any resource from the original Java world using the `getResourceContent` method.

```
public static void bar(byte[] path, int offset, int length) {
    NativeInterface ni = HIL.getInstance();
    ni.refreshContent(path, offset, length);
    String pathStr = new String(path, offset, length);
    byte[] data = ni.getResourceContent(pathStr);
    ...
}
```

Figure 11.7. *GetResourceContent Example*

11.3.3.5 Synchronous Terminations

To terminate the whole simulation (SimJPF & HIL), use the `stop()` method.

```
public static void windowClosed() {
    HIL.getInstance().stop();
}
```

Figure 11.8. *SimJPF Stop Example*

11.3.4 Shielded Plug Mock

11.3.4.1 General Architecture

The Shielded Plug Mock simulates a Shielded Plug [SP] on desktop computer. This mock can be accessed from the SimJPF, the hardware platform or a Java J2SE application.

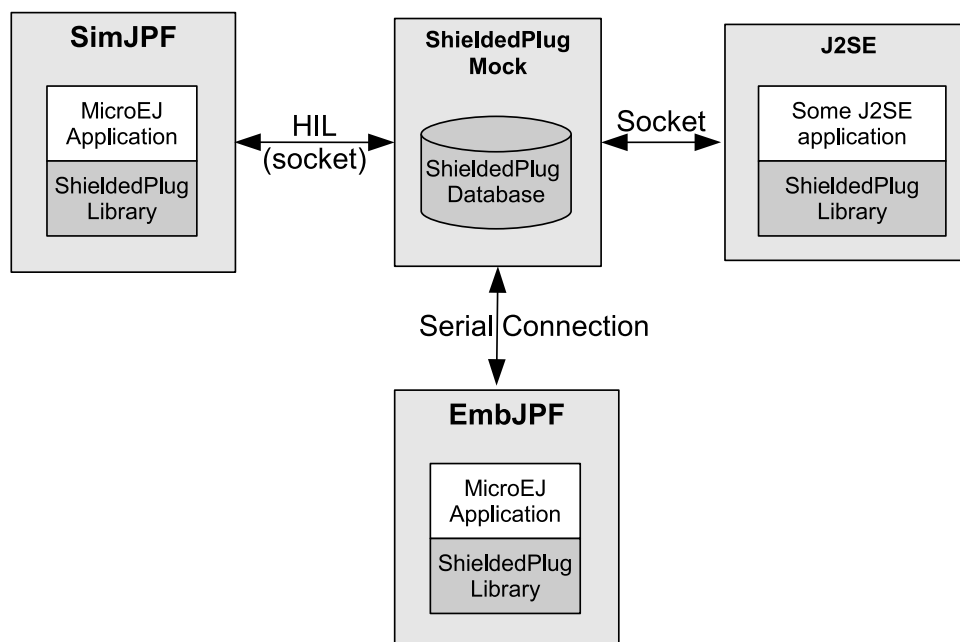


Figure 11.9. *Shielded Plug Mock General Architecture*

11.3.4.2 Configuration

The mock socket port can be customized for J2SE clients, even though several Shielded Plug mocks with the same socket port cannot run at the same time. The default socket port is 10082.

The Shielded Plug mock is a standard Java application. It can be configured using Java properties:

- `sp.connection.address`
- `sp.connection.port`

11.3.5 Dependencies

The Java platform architecture provides some APIs (HIL APIs) to develop a mock ready to be used against the simulator. The classpath variable that allows to access to the HIL Engine API is `HILENGINE-2.0`. Java projects that build mocks should put that library on their build path.

11.3.6 Installation

The mock creator is responsible to build the mock jar file using his/her own way (Eclipse build, `javac`, etc.).

Once built, the jar file must be put in this specific platform configuration project folder in order to be included during the platform creation: `dropins/mocks/dropins/`.

- First `dropins` folder: this folder has to contain all additional files and folders which will be copied into the platform as is.
- `mocks` folder: this folder has to contain all mocks jar files.
- Second `dropins` folder: this folder has to contain all miscellaneous mocks.

11.3.7 Use

Once installed, a mock is used automatically by the simulator when the Java application calls a native method which is implemented into the mock.

11.4 Dependencies

No dependency.

11.5 Installation

The simulator is a Java platform architecture built-in feature.

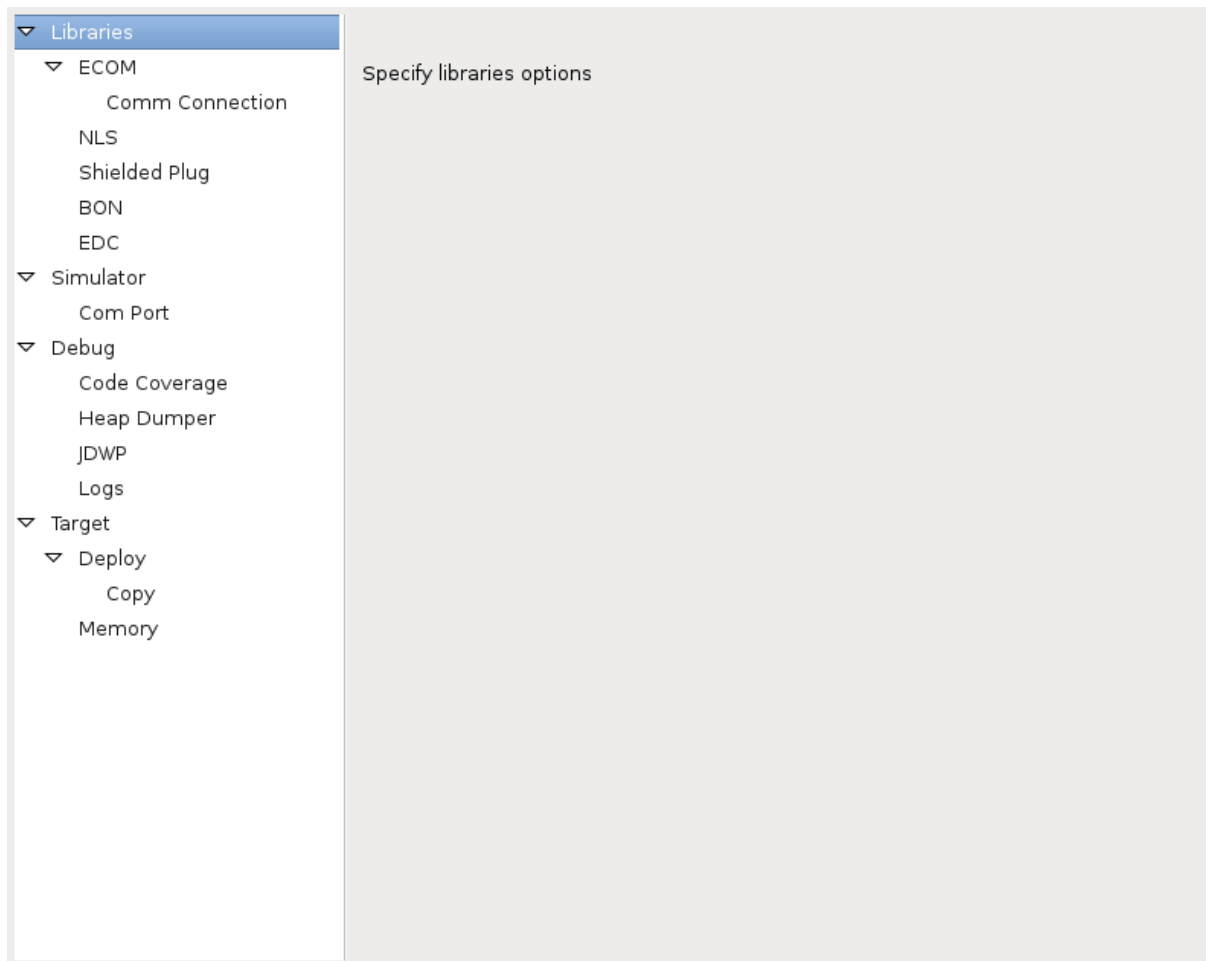
11.6 Use

To run an application in the simulator, create a MicroEJ launch configuration by right-clicking on the main class of the application and selecting **Run As** → **MicroEJ Application**.

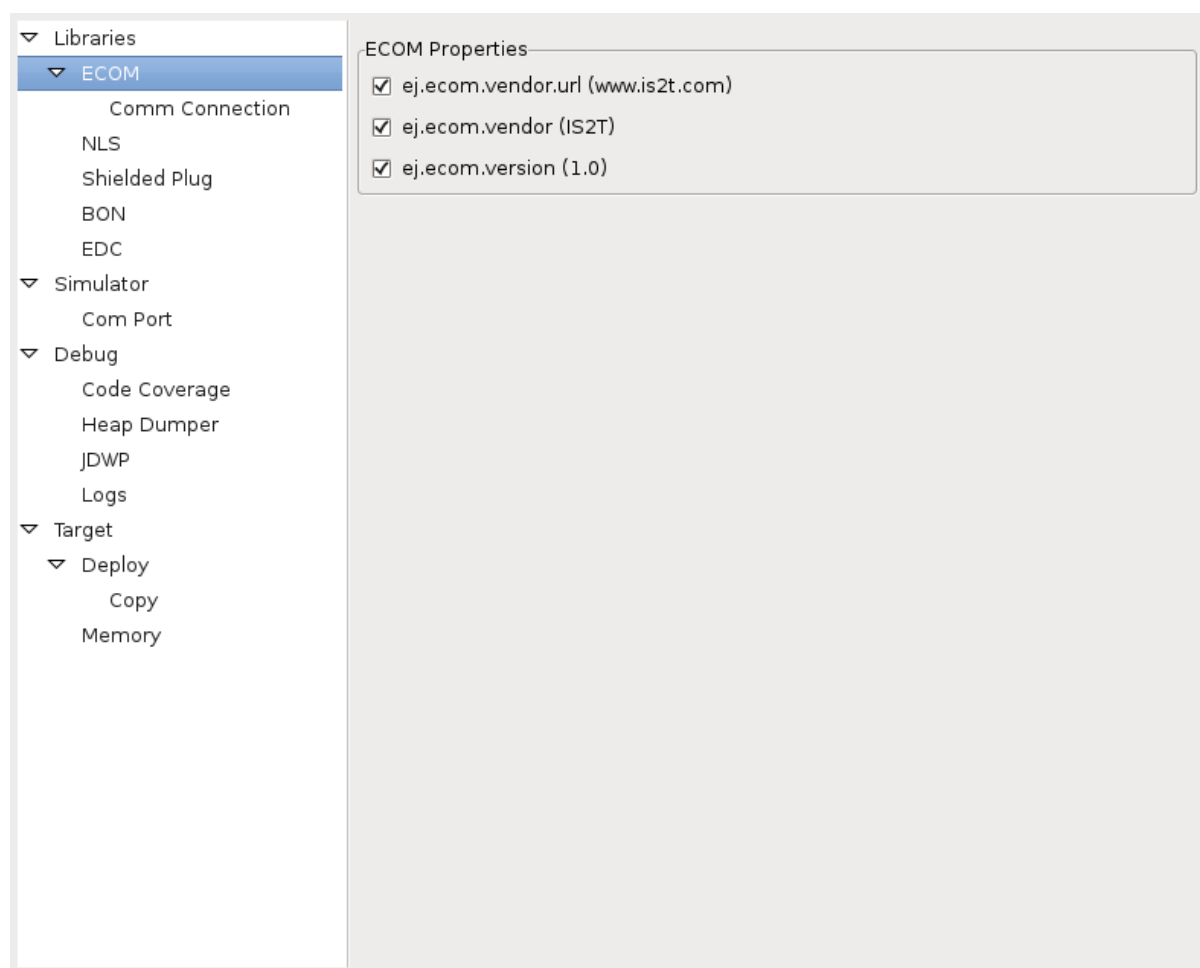
This will create a launch configuration configured for the simulator, and will run it.

12 Launch Options

12.1 Category: Libraries



12.1.1.1 Category: ECOM



12.1.1.1.1 Group: ECOM Properties

12.1.1.1.1.1 Option(checkbox): ej.ecom.vendor.url (www.is2t.com)

Default value: checked

Description:

Adds the Java property `ej.ecom.vendor.url`.

12.1.1.1.1.2 Option(checkbox): ej.ecom.vendor (IS2T)

Default value: checked

Description:

Adds the Java property `ej.ecom.vendor`.

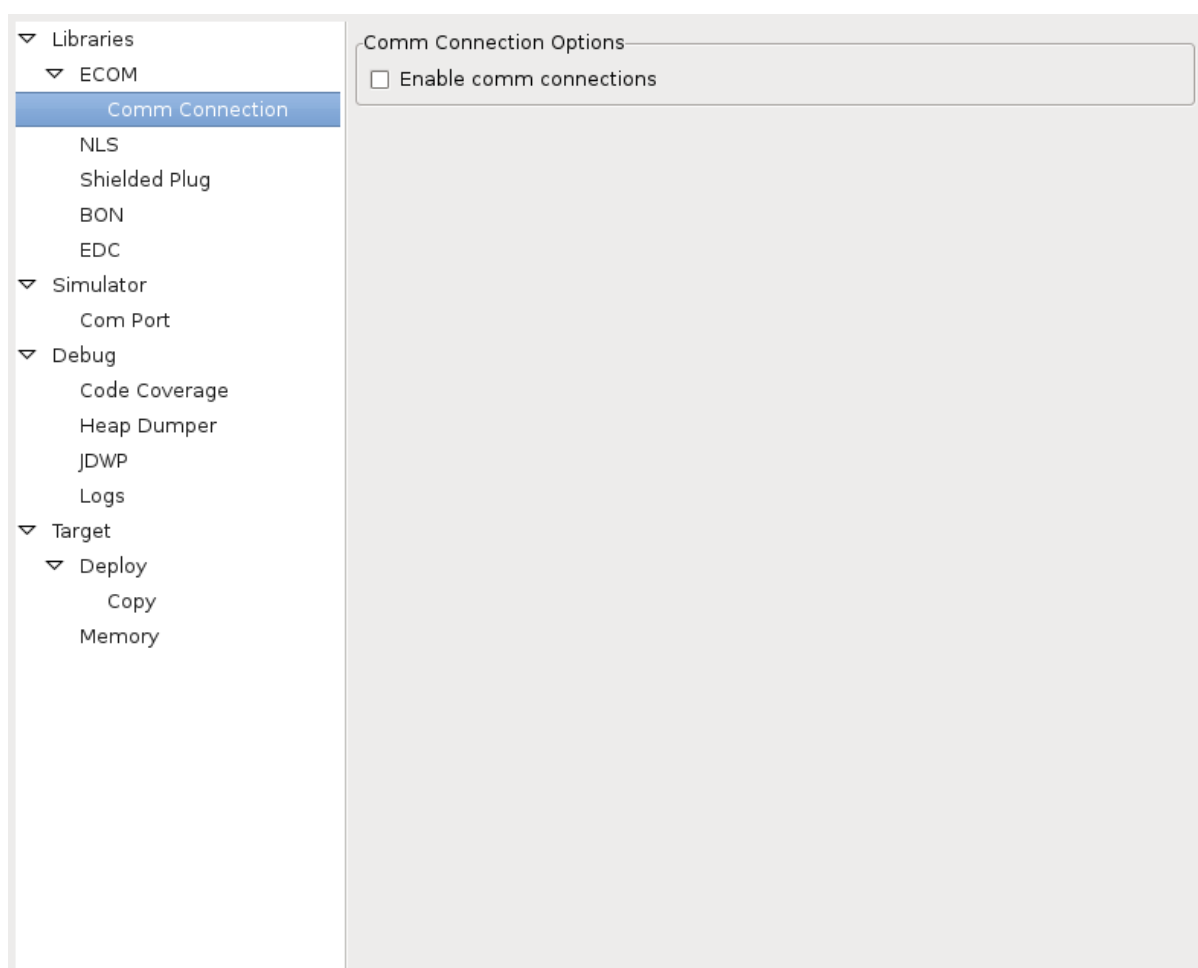
12.1.1.1.1.3 Option(checkbox): ej.ecom.version (1.0)

Default value: checked

Description:

Adds the Java property `ej.ecom.version`.

12.1.1.2 Category: Comm Connection



12.1.1.2.1 Group: Comm Connection Options

Description:

This group allows comm connections to be enabled and physical-logical mappings set. (ECOM_COMM_02)

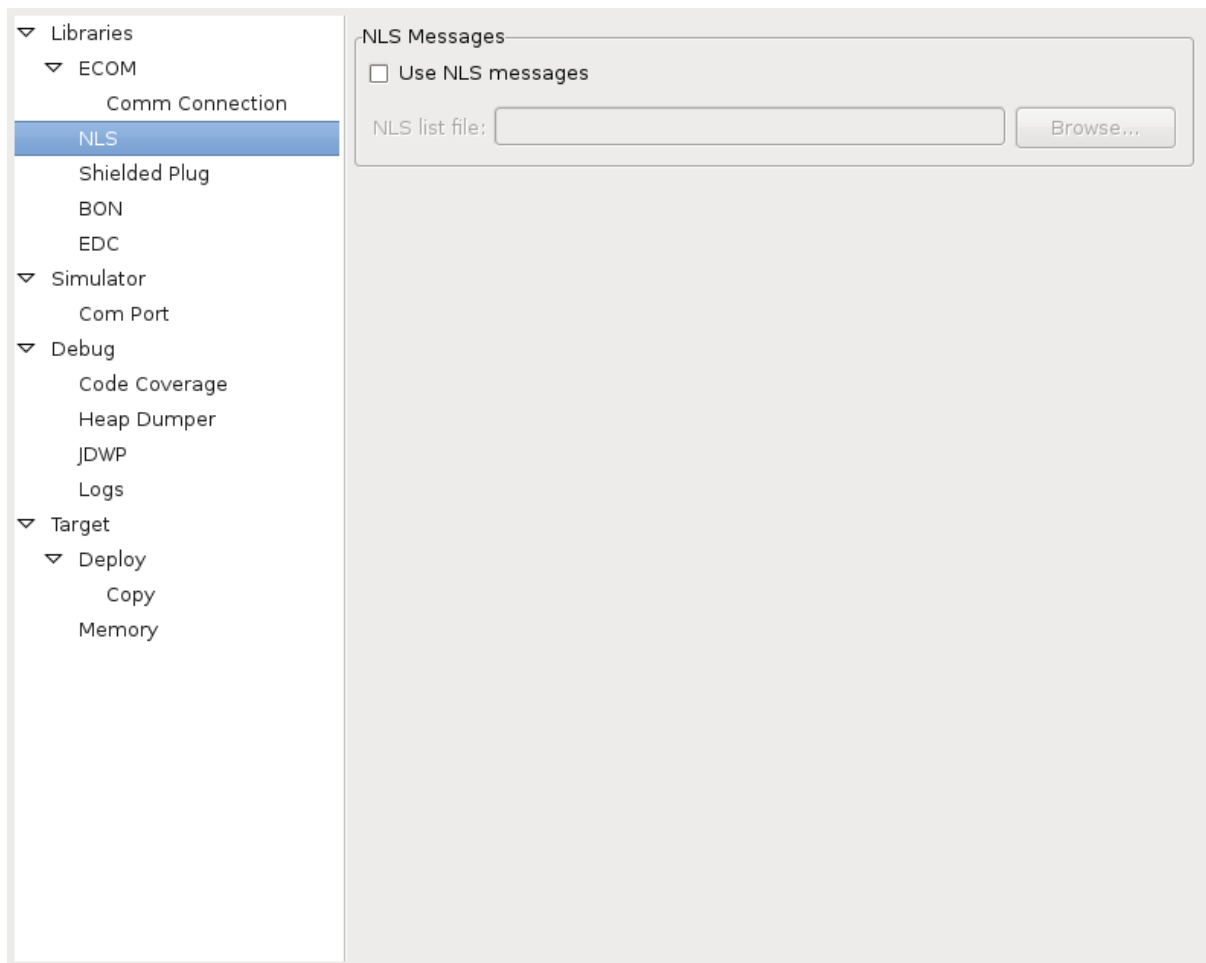
12.1.1.2.1.1 Option(checkbox): Enable comm connections

Default value: unchecked

Description:

When checked application is able to open a comm connection on an UART port. (ECOM_COMM_01)

12.1.2 Category: NLS



12.1.2.1 Group: NLS Messages

Description:

This group allows to select a file describing the NLS message which will be converted into the EmbJPF format.

12.1.2.1.1 Option(checkbox): Use NLS messages

Default value: unchecked

Description:

When selected, enables the next option NLS list file file.

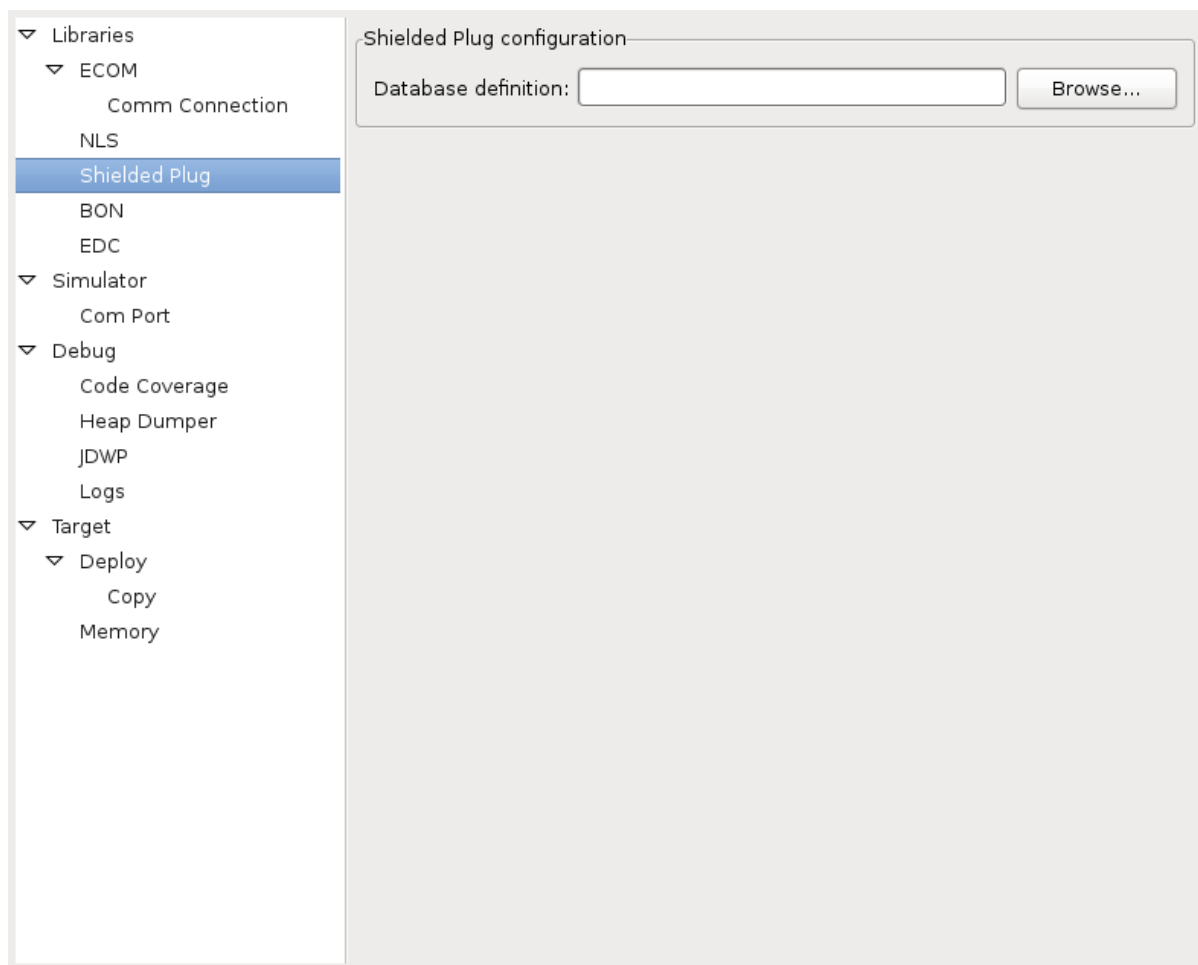
12.1.2.1.2 Option(browse): NLS list file

Default value: (empty)

Description:

Browse to select an NLS list file. Refer to NLS chapter for more information about the NLF list file format.

12.1.3 Category: Shielded Plug



12.1.3.1 Group: Shielded Plug configuration

Description:

Choose the database XML definition.

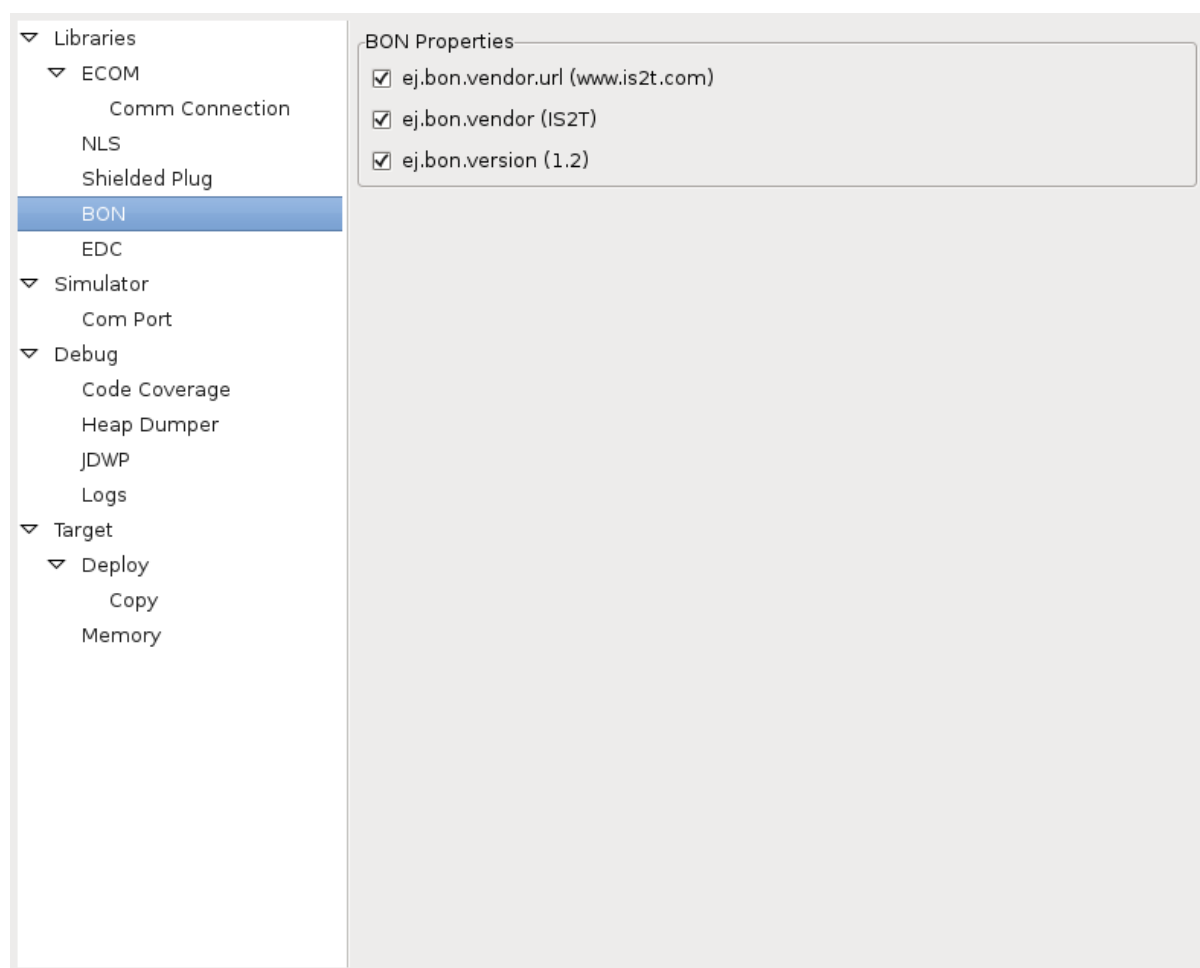
12.1.3.1.1 Option(browse): Database definition

Default value: (empty)

Description:

Choose the database XML definition.

12.1.4 Category: BON



12.1.4.1 Group: BON Properties

12.1.4.1.1 Option(checkbox): ej.bon.vendor.url (www.is2t.com)

Default value: checked

Description:

Adds the Java property `ej.bon.vendor.url`.

12.1.4.1.2 Option(checkbox): ej.bon.vendor (IS2T)

Default value: checked

Description:

Adds the Java property `ej.bon.vendor`.

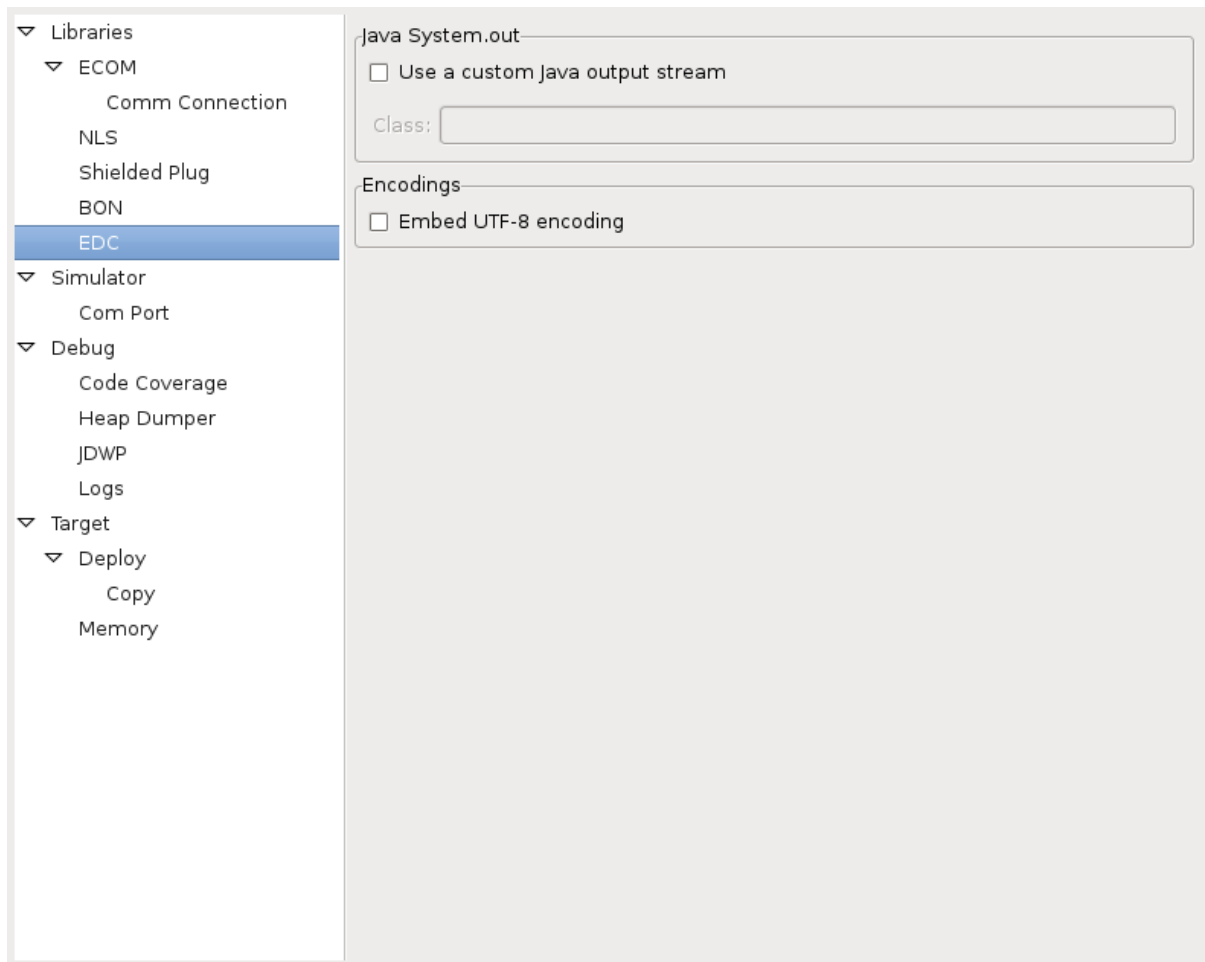
12.1.4.1.3 Option(checkbox): ej.bon.version (1.2)

Default value: checked

Description:

Adds the Java property `ej.bon.version`.

12.1.5 Category: EDC



12.1.5.1 Group: Java System.out

12.1.5.1.1 Option(checkbox): Use a custom Java output stream

Default value: unchecked

Description:

Select this option to specify another Java System.out print stream (CORE_01).

If selected, the default Java output stream is not used by the Java application. the JPF will not use the default Java output stream at startup.

12.1.5.1.2 Option(text): Class

Default value: (empty)

Description:

Format: Java class like packageA.packageB.className

Defines the Java class used to manage System.out.

At startup the JPF will try to load this class using the `Class.forName()` method. If the given class is not available, the JPF will use the default Java output stream as usual. The specified class must be available in the application classpath.

12.1.5.2 Group: Encodings

Description:

Specifies the encodings to embed.

12.1.5.2.1 Option(checkbox): Embed UTF-8 encoding

Default value: unchecked

Description:

Embed UTF-8 encoding (CORE_02).

12.2 Category: Simulator

The screenshot shows the Keil uVision Simulator configuration interface. On the left, a tree view lists various categories: Libraries, ECOM, Comm Connection, NLS, Shielded Plug, BON, EDC, Simulator (selected), Com Port, Debug, Code Coverage, Heap Dumper, JDWP, Logs, Target, Deploy, Copy, and Memory. The main configuration area on the right is divided into three sections:

- Shielded Plug server configuration:** Contains a text input field for 'Server socket port' with the value 10082.
- Options:** Contains a checkbox for 'Use target characteristics' which is unchecked, and a text input field for 'Slowing factor (0 means deactivated):' with the value 0.
- HIL Connection:** Contains a checkbox for 'Specify a port' which is unchecked, a text input field for 'HIL connection port:' with the value 8001, and a text input field for 'HIL connection timeout:' with the value 10.

12.2.1 Group: Shielded Plug server configuration

Description:

This group allows configuration of the Shielded Plug database.

12.2.1.1 Option(text): Server socket port

Default value: 10082

Description:

Set the Shielded Plug server socket port.

12.2.2 Group: Options

Description:

This group specifies options for SimJPF.

12.2.2.1 Option(checkbox): Use target characteristics

Default value: unchecked

Description:

When selected, this option forces the SimJPF to use the EmbJPF exact characteristics (SIMJPF_03). It sets the SimJPF scheduling policy according to the EmbJPF one. It forces resources to be explicitly specified. It enables log trace and gives information about the RAM memory size the EmbJPF uses.

12.2.2.2 Option(text): Slowing factor (0 means deactivated)

Default value: 0

Description:

Format: Positive integer

This option allows the SimJPF to be slowed down in order to match the EmbJPF execution speed. The greater the slowing factor, the slower the SimJPF runs (SIMJPF_04).

12.2.3 Group: HIL Connection

Description:

This group enables the control of HIL (Hardware In the Loop) connection parameters (connection between SimJPF and the mocks).

12.2.3.1 Option(checkbox): Specify a port

Default value: unchecked

Description:

When selected allows the use of a specific HIL connection port, otherwise a random free port is used.

12.2.3.2 Option(text): HIL connection port

Default value: 8001

Description:

Format: Positive integer

Values: [1024-65535]

It specifies the port used by the SimJPF to accept HIL connections.

12.2.3.3 Option(text): HIL connection timeout

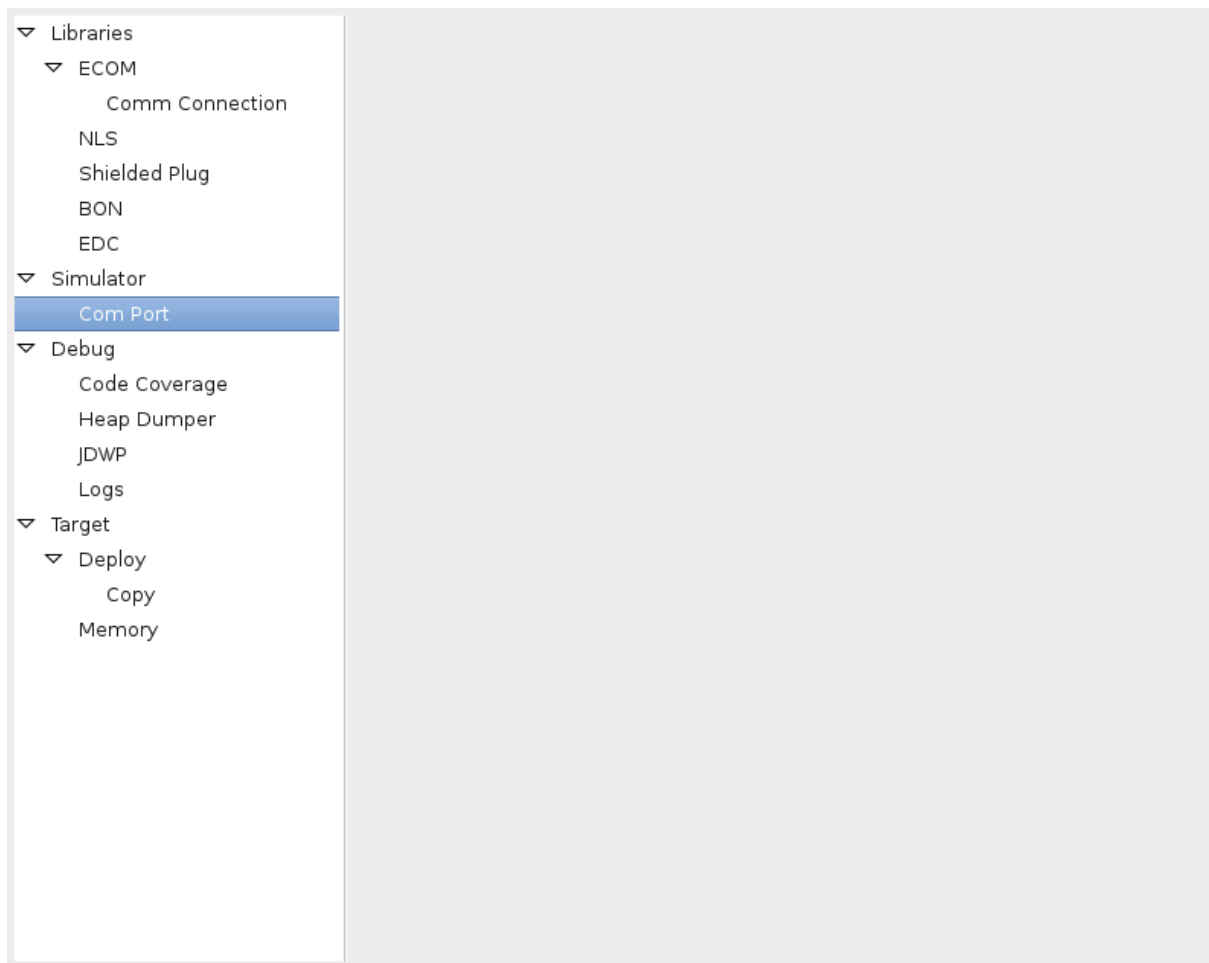
Default value: 10

Description:

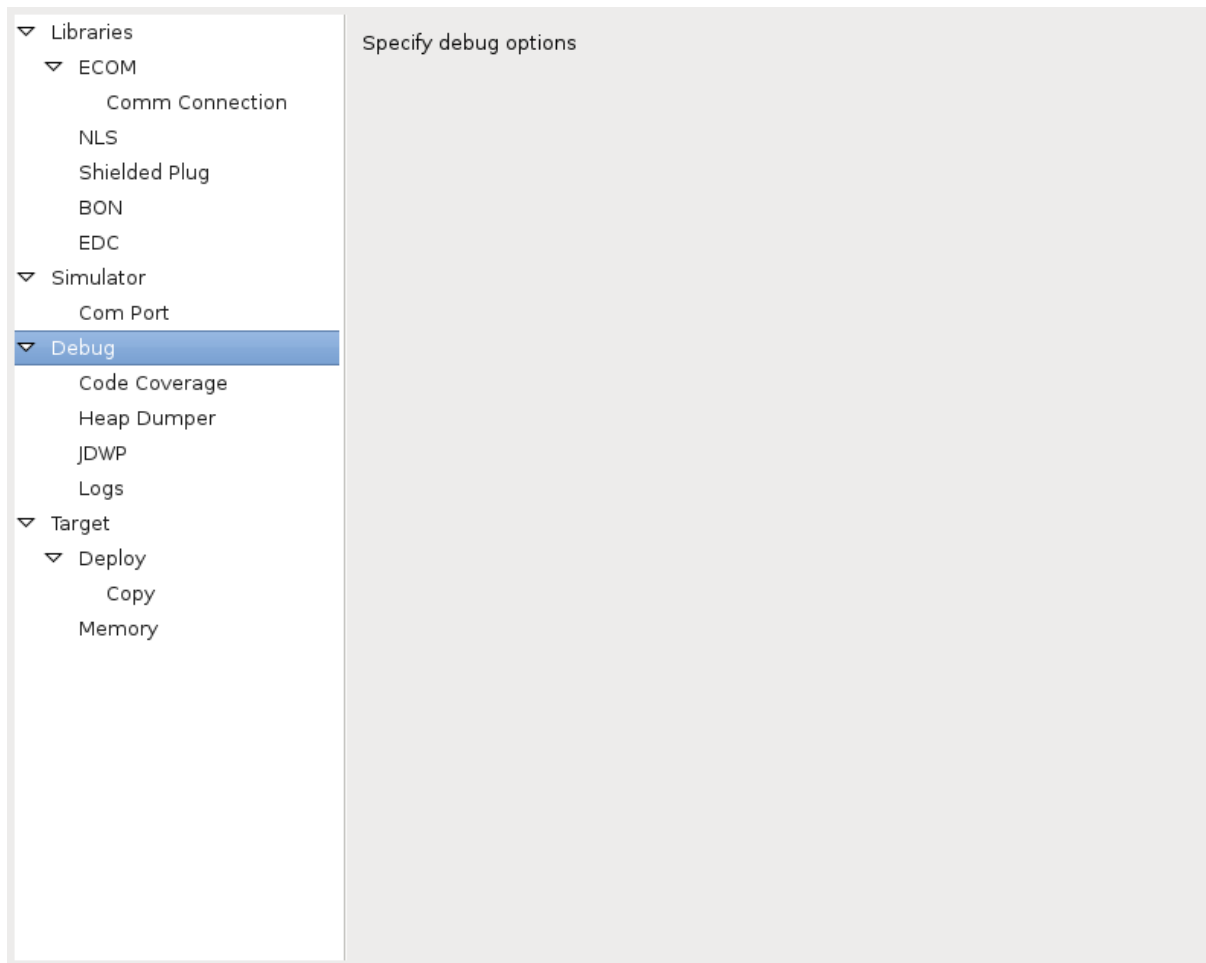
Format: Positive integer

It specifies the time the SimJPF should wait before failing when it invokes native methods.

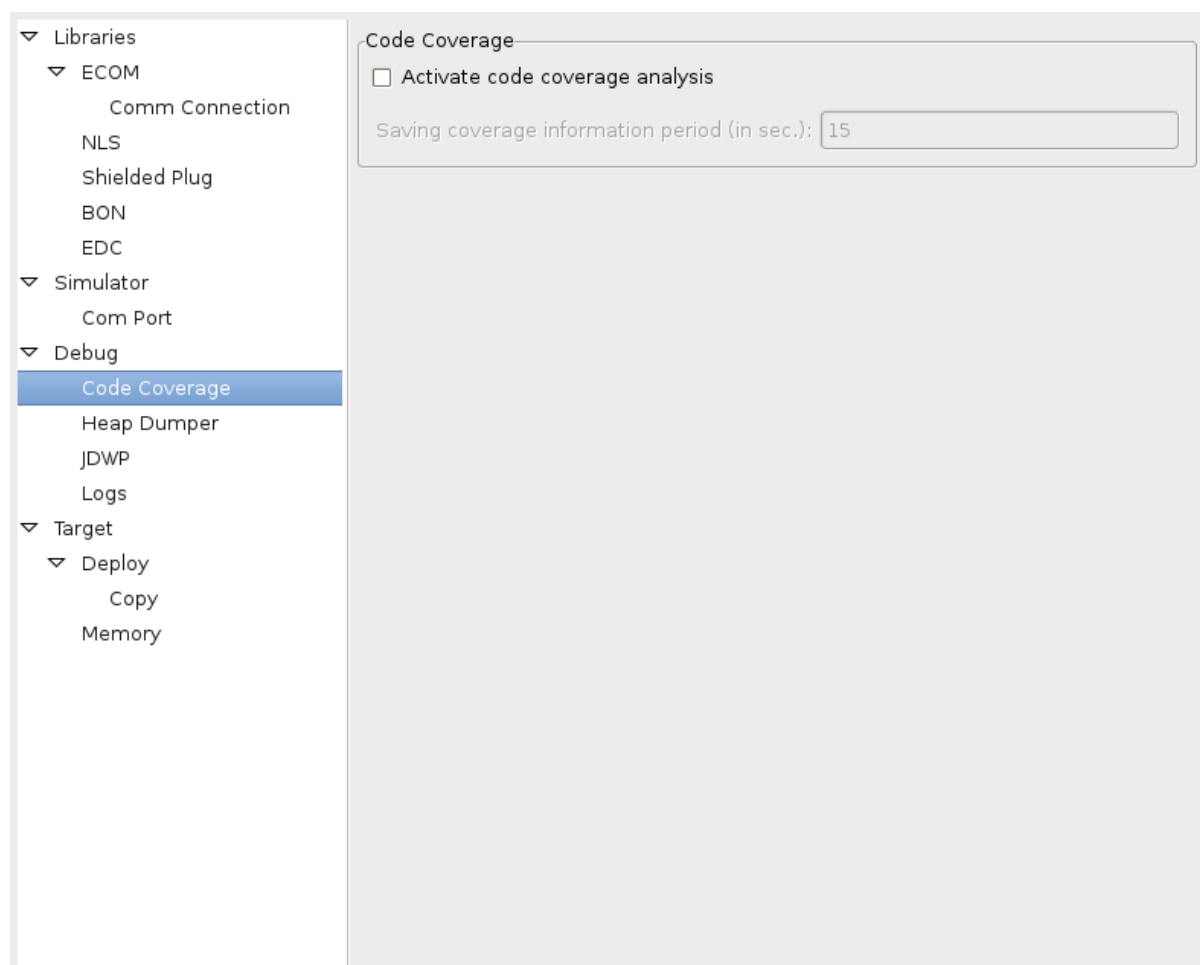
12.2.4 Category: Com Port



12.3 Category: *Debug*



12.3.1 Category: Code Coverage



12.3.1.1 Group: Code Coverage

Description:

This group is used to set parameters of the code coverage analysis tool (SIMJPF_05).

12.3.1.1.1 Option(checkbox): Activate code coverage analysis

Default value: unchecked

Description:

When selected it enables the code coverage analysis by the SimJPF. Resulting files are output in the cc directory inside the output directory.

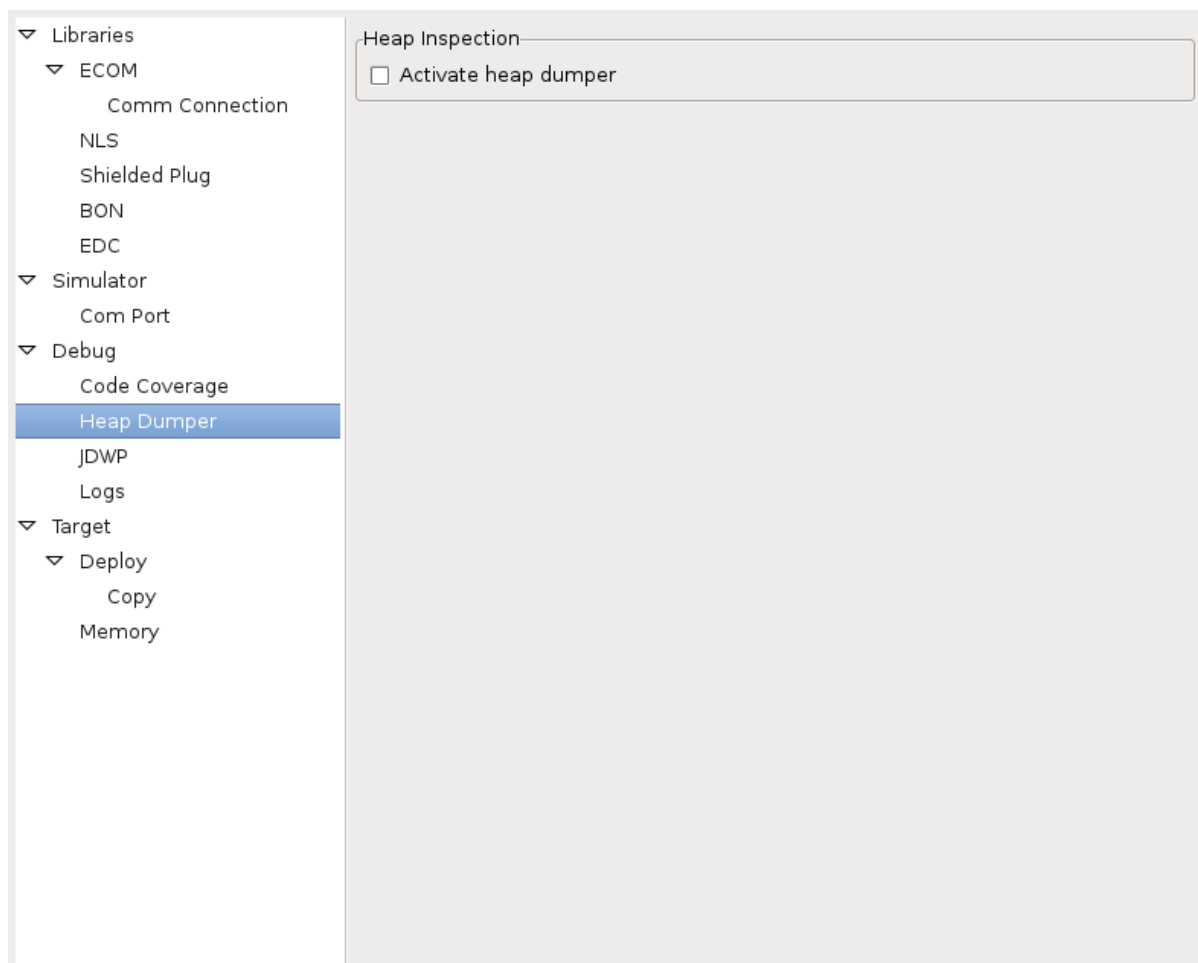
12.3.1.1.2 Option(text): Saving coverage information period (in sec.)

Default value: 15

Description:

It specifies the period between the generation of .cc files.

12.3.2 Category: Heap Dumper



12.3.2.1 Group: Heap Inspection

Description:

This group is used to specify heap inspection properties.

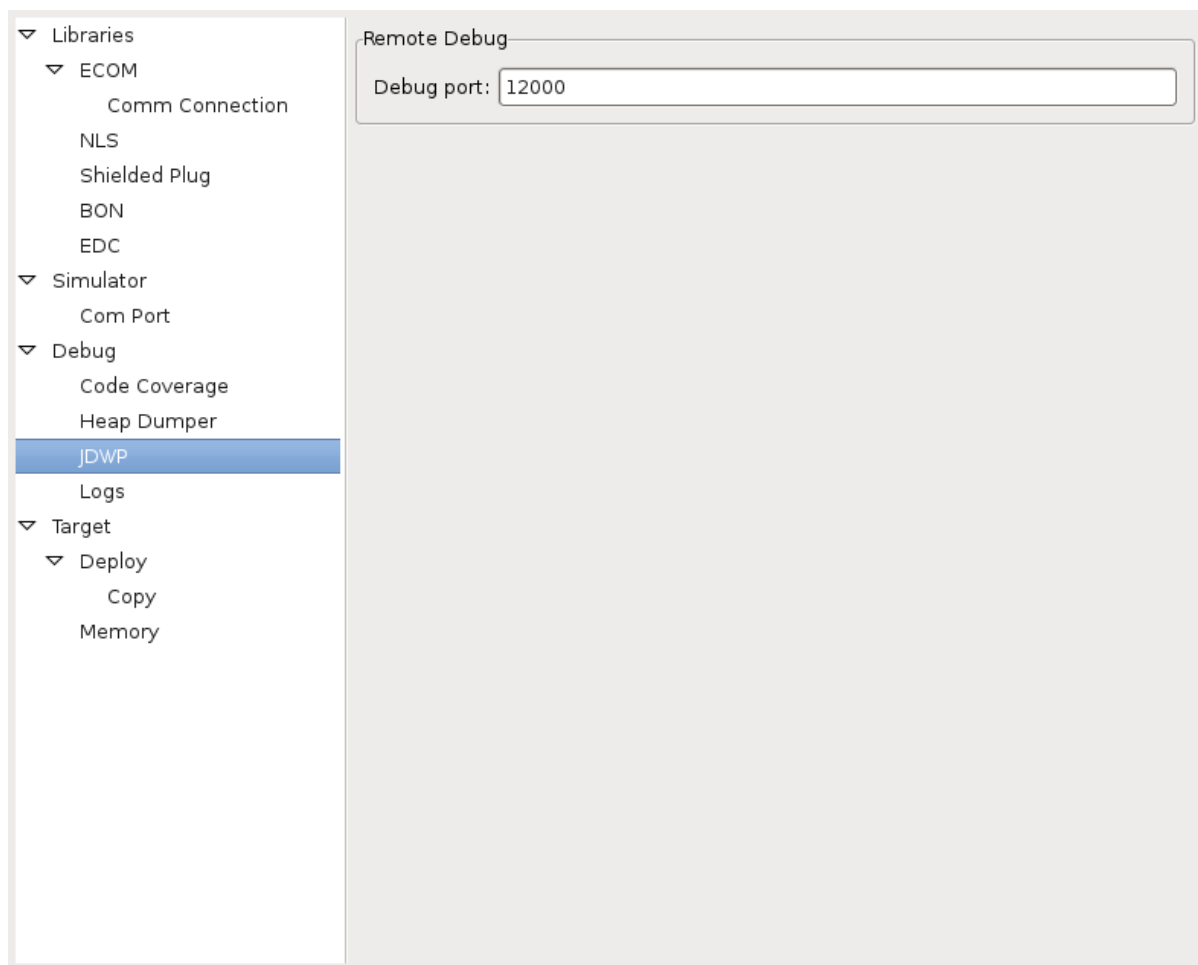
12.3.2.1.1 Option(checkbox): Activate heap dumper

Default value: unchecked

Description:

When selected, this option enables a dump of the heap each time the `System.gc()` method is called by the Java application (SIMJPF_02).

12.3.3 Category: JDWP



12.3.3.1 Group: Remote Debug

12.3.3.1.1 Option(text): Debug port

Default value: 12000

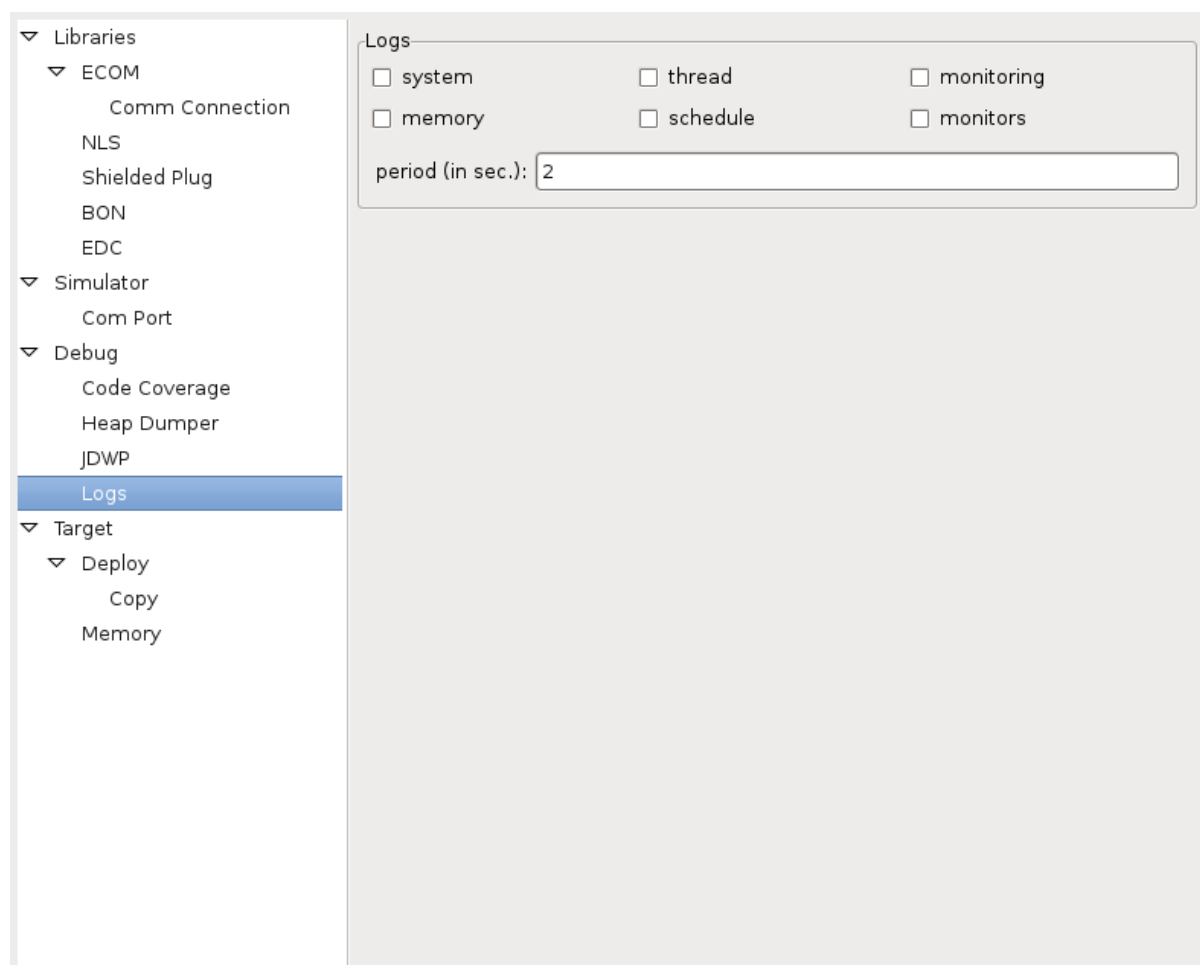
Description:

Configures the JDWP debug port (SIMJPF_01).

Format: Positive integer

Values: [1024-65535]

12.3.4 Category: Logs



12.3.4.1 Group: Logs

Description:

This group defines parameters for SimJPF log activity. Note that logs can only be generated if the Simulator > Use target characteristics option is selected.

Some logs are sent when the JPF executes some specific action (such as start thread, start GC, etc), other logs are sent periodically (according to defined log level and the log periodicity).

12.3.4.1.1 Option(checkbox): system

Default value: unchecked

Description:

When selected, System logs are sent when the JPF executes the following actions:

start and terminate a Java thread

start and terminate a GC

exit

12.3.4.1.2 Option(checkbox): thread

Default value: unchecked

Description:

When selected, thread information is sent periodically. It gives information about alive Java threads (status, memory allocation, stack size).

12.3.4.1.3 Option(checkbox): monitoring

Default value: unchecked

Description:

When selected, thread monitoring logs are sent periodically. It gives information about time execution of Java threads.

12.3.4.1.4 Option(checkbox): memory

Default value: unchecked

Description:

When selected, memory allocation logs are sent periodically. This level allows to supervise memory allocation.

12.3.4.1.5 Option(checkbox): schedule

Default value: unchecked

Description:

When selected, a log is sent when the JPF schedules a Java thread.

12.3.4.1.6 Option(checkbox): monitors

Default value: unchecked

Description:

When selected, monitors information is sent periodically. This level permits tracing of all thread state by tracing monitor operations.

12.3.4.1.7 Option(text): period (in sec.)

Default value: 2

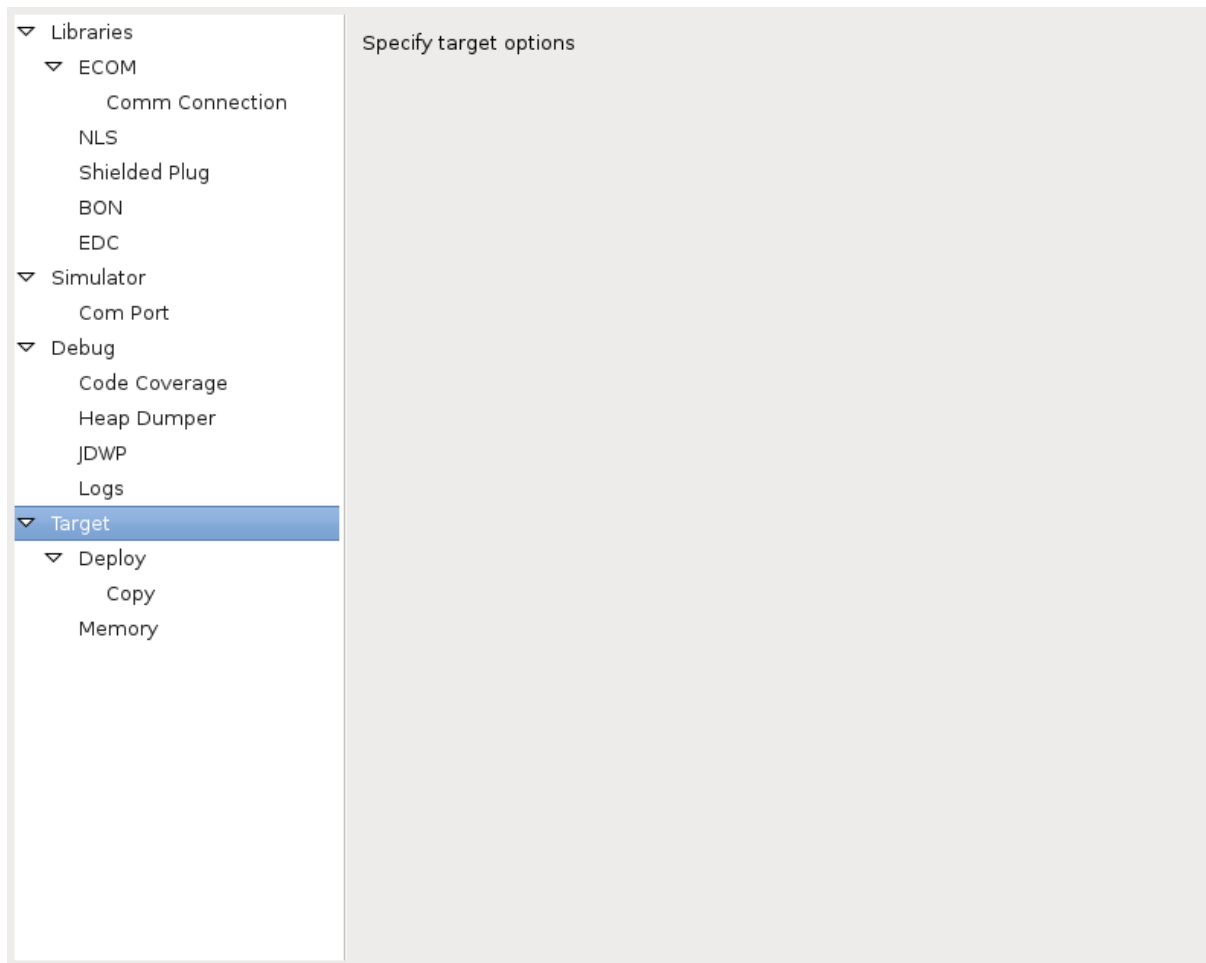
Description:

Format: Positive integer

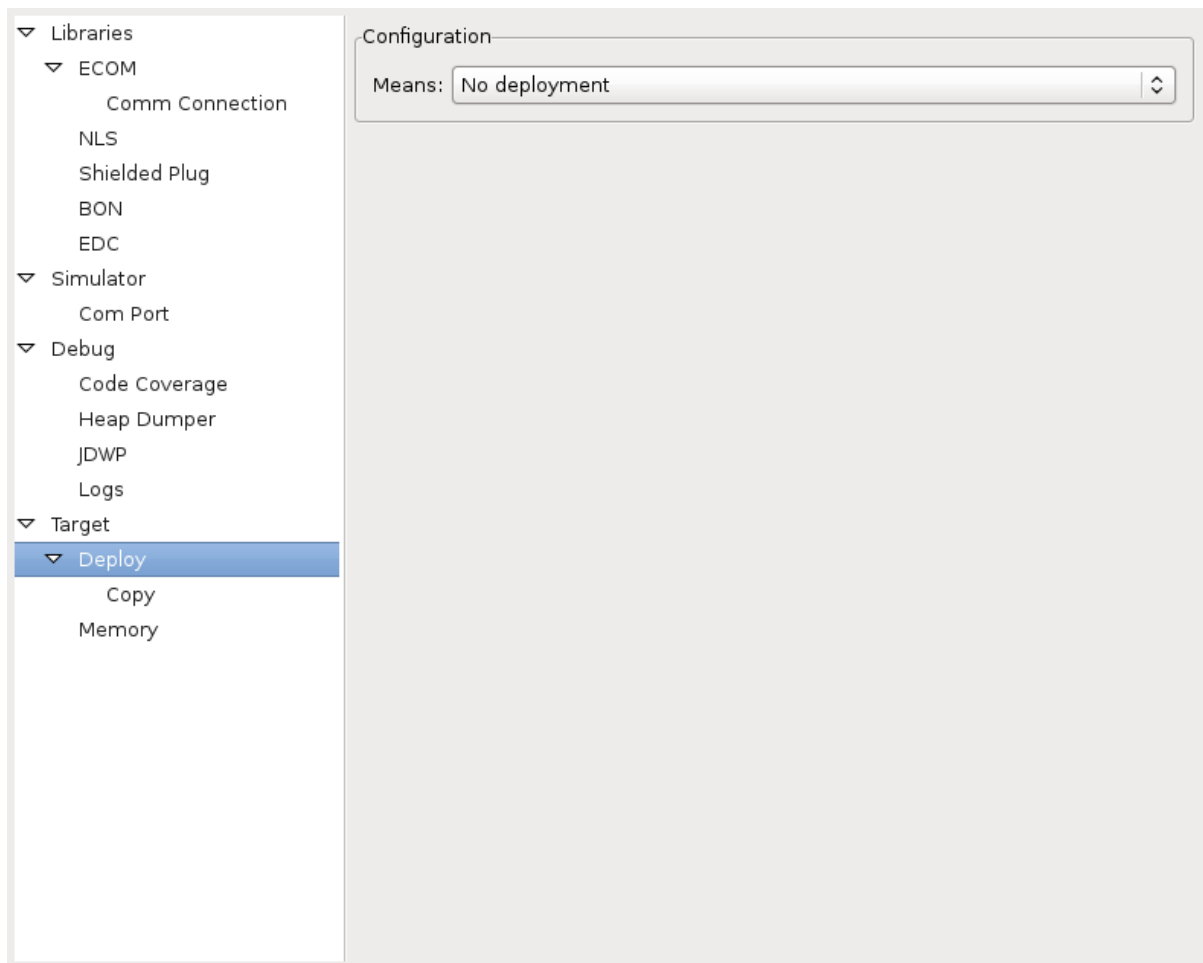
Values: [0-60]

Defines the periodicity of periodical logs.

12.4 Category: Target



12.4.1 Category: Deploy



12.4.1.1 Group: Configuration

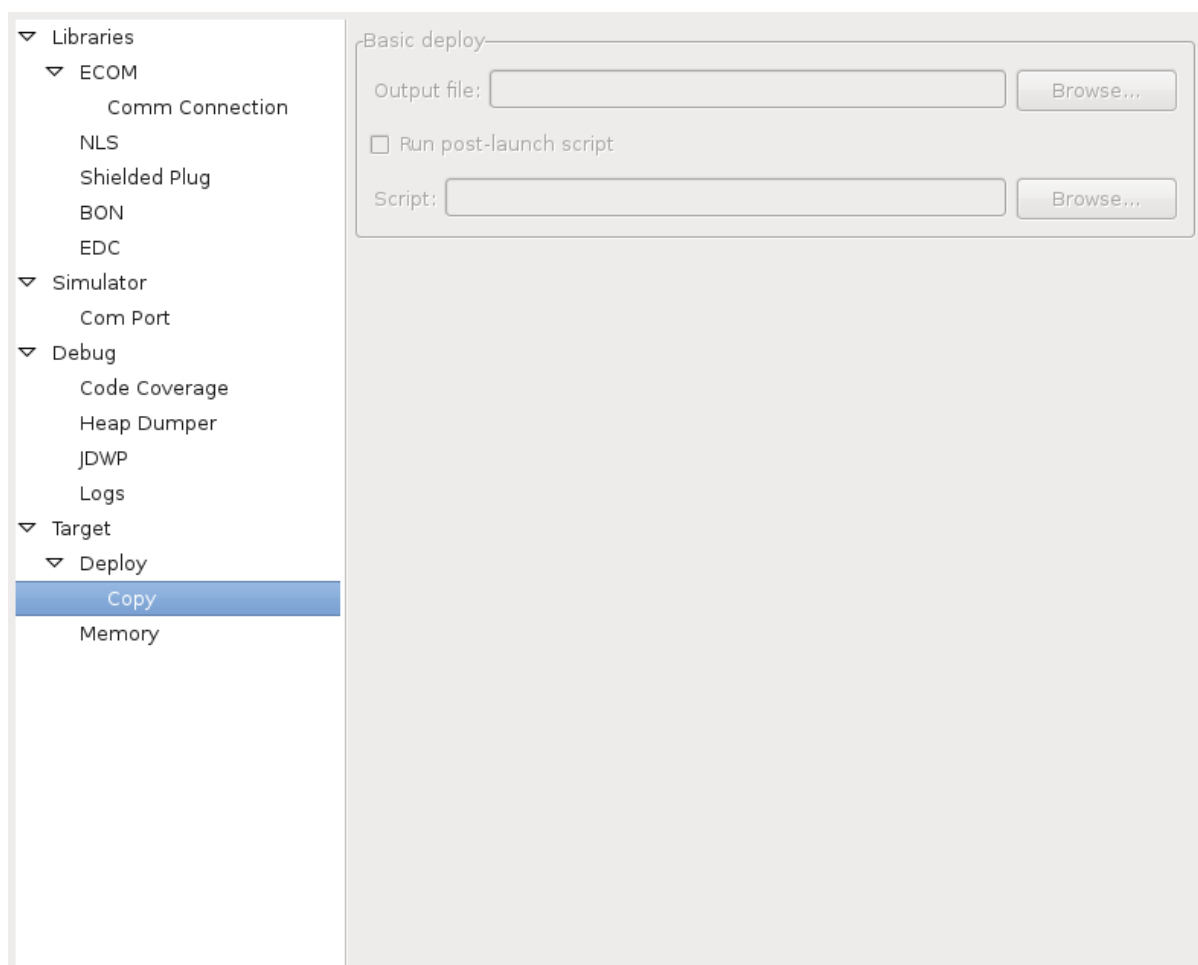
12.4.1.1.1 Option(combo): Means

Default value: No deployment

Available values:

No deployment

Copy

12.4.1.2 Category: Copy*Description:*

Choose an output folder to hold the compiled java application.

12.4.1.2.1 Group: Basic deploy**12.4.1.2.1.1 Option(browse): Output file**

Default value: (empty)

Description:

Select an output file.

12.4.1.2.1.2 Option(checkbox): Run post-launch script

Default value: unchecked

Description:

Enable to run a post-launch script for application specific deployment.

12.4.1.2.1.3 Option(browse): Script

Default value: (empty)

Description:

Choose ANT script to be run after launch.

12.4.2 Category: Memory

The screenshot shows the STM32Java configuration window. On the left, a tree view under 'Target' has 'Memory' selected. The main area is divided into two sections: 'Heaps' and 'Threads'. The 'Heaps' section contains two input fields: 'java heap size (in bytes)' with the value 32768 and 'Immortal heap size (in bytes)' with the value 4096. The 'Threads' section contains four input fields: 'Number of threads' with the value 5, 'Number of blocks in pool' with the value 10, 'Block size (in bytes)' with the value 512, and 'Maximum size of thread stack (in blocks)' with the value 2.

Category	Option	Value
Heaps	java heap size (in bytes)	32768
	Immortal heap size (in bytes)	4096
Threads	Number of threads	5
	Number of blocks in pool	10
	Block size (in bytes)	512
	Maximum size of thread stack (in blocks)	2

12.4.2.1 Group: Heaps

12.4.2.1.1 Option(text): Java heap size (in bytes)

Default value: 32768

Description:

Specifies the Java heap size in bytes (JPF_01).

A Java heap contains live Java objects. An OutOfMemory error can occur if the heap is too small.

12.4.2.1.2 Option(text): Immortal heap size (in bytes)

Default value: 4096

Description:

Specifies the Java Immortal heap size in bytes (BON_02).

The Java Immortal heap contains allocated Java Immortal objects. An OutOfMemory error can occur if the heap is too small.

12.4.2.2 Group: Threads

Description:

This group allows the configuration of application and library Java thread(s). A Java thread needs a stack to run. This stack is allocated from a pool and this pool contains several blocks. Each block has the same size. At thread startup the thread uses only one block for its stack. When the first block is full it uses another block. The maximum number of blocks per thread must be specified. When the maximum number of blocks for a thread is reached or when there is no free block in the pool, a StackOverflow error is thrown. When a thread terminates all associated blocks are freed. These blocks can then be used by other threads.

12.4.2.2.1 Option(text): Number of threads

Default value: 5

Description:

Specifies the number of threads the application will be able to use at the same time (JPF_02).

12.4.2.2.2 Option(text): Number of blocks in pool

Default value: 10

Description:

Specifies the number of blocks in the stacks pool (JPF_03).

12.4.2.2.3 Option(text):

Default value: 512

Description:

Specifies the thread stack block size (in bytes).

12.4.2.2.4 Option(text):

Default value: 2

Description:

Specifies the maximum number of blocks a thread can use (JPF_04). If a thread requires more blocks a StackOverflow error will occur.

13 Appendix

13.1 Keil uVision Compiler Compatibility

The Java platform architecture is compatible from Keil uVision 4.x to Keil uVision 5.1.

13.2 Floating Point Unit

The ARM Cortex-M4 processor has a number of features that the ARM Cortex-M3 does not. This includes an optional floating point unit (FPU). This FPU is single precision (32 bits) and is compliant to IEEE 754 standard. It can be disabled when not in use, reducing power consumption.

There are two steps to use the FPU in an application. The first step is to tell the compiler and the linker that the microcontroller has a FPU available so that they will produce compatible binary code. The second step is to enable the FPU during execution. This is done by writing to CPAR in the `SystemInit()` function.

Even if you have a FPU in the processor, you may still need to use run-time library functions to deal with advanced operations. A program may also defines calculations functions with floating numbers, either as parameter or return value. There are several Application Binary Interfaces (ABI) to handle floating point calculations. Hence, most compilers provide options to select one of these ABIs. This will affect how parameters are passed between caller functions and callee functions, and even if the FPU is used or not. There are 3 different ABIs:

- Soft ABI without FPU hardware. Values are passed via integer registers.
- Soft ABI with FPU hardware. FPU is accessed directly for simple operations but when a function is called, the integer registers are used.
- Hard ABI. FPU is accessed directly for simple operations and FPU-specific registers are used when a function is called, for both parameters and return value.

It is important to notice that code compiled with a particular ABI may or may not be compatible with code compiled with another ABI.

MicroEJ modules, included the MicroJvm virtual machine, use the soft ABI with FPU hardware. Hence, in C project:

- the soft ABI with FPU hardware will ensure the best compatibility,
- the soft ABI without FPU hardware since function calling conventions are compatible,
- the Hard ABI is forbidden because function calling conventions are different.

14 Document History

Date	Revision	Description
October 24th 2012	A	First release
November 20th 2013	B	Global documentation review
June 20th 2014	C	Update for STM32Java 3.0.0